プロセスのふるまいに基づく計算機資源管理

プロセスのふるまいに基づく計算機資源管理

2017年3月

中川　岳

プロセスのふるまいに基づく計算機資源管理

中川　岳

システム情報工学研究科
筑波大学

2017年3月

概要

プログラムの実行インスタンスであるプロセスは，その動作のために CPU 時間，メモリ，ストレージといった種々のリソースを消費する．計算機システムにおけるリソースは有限であり，プロセスそれぞれのリソース利用状況に基づいて管理が行われる．本研究ではそのリソースの利用状況に加えて，リソース利用の経過情報をその管理に利用することを議論する．

プロセスのリソース利用の経過情報はリソース管理に有用である．例えば，複数のリソース利用量とタイミングの情報から，そのプロセスがリソースを消費する速度を得ることができる．この速度を用いる事で，ある時点に対して次の瞬間のリソースの消費状況を予測することができる．本研究では，このリソース利用の経過情報から得られるリソース利用の傾向をリソース利用のふるまいと定義し，これを活用して既存のリソース管理手法の問題を解決することを試みる．

本研究では，計算機システムにおけるリソース管理の中でも，特にメモリリソースの管理に着目する．計算機システムおけるメモリ資源は有限であり，システムの安定動作や高性能化のためには，その割り当てやデータ配置を適切に管理することが求められる．これまで，その管理は直近のメモリ利用状況に基づいて実施されてきた．しかしながら，サーバシステムの構成の変化や不揮発性メモリによるメモリ階層の複雑化に伴い，この直近の利用状況のみに基づいたメモリリソースの管理だけでは，メモリ管理に関する問題を防いだり，メモリシステムの性能を十分に引き出すことができなくなる．

本研究では，前述した問題の解決にプロセスのメモリリソース利用のふるまいを活用することを試みる．例えば単位時間あたりのメモリ使用量の増分や，データの意味によって異なるメモリへの書き込みアクセスの回数などがメモリリソース利用のふるまいである．このふるまいを利用することで，プロセスのメモリリソース利用の傾向を予測することができる．

本研究では，このメモリリソース消費のふるまいを利用したメモリ管理の手法として，二つの手法を提案する．一つは，不揮発性メモリと DRAM によって構成されるハイブリッドメモリアーキテクチャにおけるデータ配置手法である．もう一つは，ソフトウェアによる異常なメモリリソースの消費を未然に防ぐ，プロセスのリソース隔離である．それぞれの提案手法により研究の背景であるメモリリソース管理を取り巻く諸問題を解決可能である．

目次

図目次

表目次

第 1 章

はじめに

プログラムの実行インスタンスであるプロセスは，その動作のために CPU 時間，メモリ，ストレージといった種々のリソースを消費する．計算機システムにおけるリソースは有限であり，プロセスそれぞれのリソース利用状況に基づいて管理が行われる．例えばメモリの使用量ならば，あるプロセスへのメモリページの割り当て時にその時点での対象プロセスのメモリ利用量を確認し，そのプロセスが利用できる上限を超えていないかを確認する．この時，それ以前のメモリリソース利用の状況は考慮されない．

しかしながら，プロセスのリソース利用の経過情報はリソース管理に有用である．例えば，複数のリソース利用量とタイミングの情報から，そのプロセスがリソースを消費する速度を得ることができる．この速度を用いる事で，次の瞬間のリソースの消費状況を予測することができる．本研究では，このリソース利用の経過情報から得られるリソース利用の傾向をリソース利用のふるまいと定義し，これを活用して既存のリソース管理手法の問題を解決することを試みる．

過去の情報に基づき将来のプログラムの挙動を予想することは，計算機システムの運用における様々な局面で行われてきた．例えばハードウェアからソフトウェアのレベルまでさまざまな機構に存在するキャッシュは，プログラムがデータにアクセスする一般的にふるまいである参照の局所性に依拠している．プロセッサが備える分岐予測と投機的実行も，プログラムのふるまいに基づいた予測を利用している．これらの先行するアイデアと同様に，本研究では，過去の情報から導かれるリソース利用のふるまいを積極的に利用することで，複雑化するリソース管理に関する諸問題を解決することを目指す．

本研究では，計算機システムにおけるリソース管理の中でも，特にメモリリソースの管理に着目する．計算機システムおけるメモリ資源は有限であり，システムの安定動作や高性能化のためには，その割り当てやデータ配置を適切に管理することが求められる．これまで，その管理は直近のメモリ利用状況に基づいて実施されてきた．しかしながら，サーバシステムの構成の変化や不揮発性メモリ（NVM: Non-Volatile Memory）によるメモリ階層の複雑化に伴い，この直近の利用状況のみに基づいたメモリリソースの管理だけでは，メモリ管理に関する問題を防いだり，メモリシステムの性能を十分に引き出すことができなくなる．

本研究は，この問題の解決にプロセスのメモリリソース利用のふるまいを活用することを試みる．

プロセスのメモリリソース利用のふるまいとは，プロセスごとに異なる，メモリ利用状況の経過情報から導かれるメモリ利用の特徴を表す情報である．例えば単位時間あたりのメモリ使用量の増分や，データの意味ごとに異なるメモリへの書き込みアクセスの回数などがメモリリソース利用のふるまいである．このふるまいを利用することで，プロセスのメモリリソース利用の傾向を予測することができる．

1.1 研究の背景

従来の直近の利用状況のみに基づいたメモリリソース管理のみでは，複雑化する計算機システムの安定運用や新しいメモリデバイスの有効利用が困難である．以下では，その背景として 2 つのことを述べる．

1.1.1 不揮発性メモリによるメモリ階層の複雑化

不揮発性メモリ（Non-volatile Memory: NVM）の技術革新により，計算機システムのメモリ階層は大きく変化する可能性がある．現在，STT-MRAM [1], ReRAM [2]，PCM [3] といった次世代の NVM 素子の研究開発が進んでいる．ストレージとして広く利用されている NAND Flash についても，積層実装によりその大容量化や高速化といった性能向上が続いている [4]．SATA や PCI-Express といった I/O バス経由でアクセスされている SSD をメモリバス経由で接続する規格も検討されている [5]．主記憶としての利用を想定した NVM モジュールの開発も行われている [6]．従来は大容量化が難しいとされていた STT-MRAM に関しても，DRAM と同等の集積度を満たしつつある [7]．取り扱うデータの規模が巨大である高性能計算（HPC）の分野でもストレージや主記憶としての利用が積極的に検討されている [8]．HPC 環境向けに NVM を有効利用するためのプログラミング言語の拡張も提案されている [9]．

現時点ではメモリ階層の中での NVM の利用形態は定まっていないが，その一つとして主記憶の拡張がある．NVM の 1 つである PCM は DRAM よりも高い記録密度を持つ．このようなメモリ素子を用いて主記憶を構成することで，従来の DRAM 主体の主記憶に比べて，主記憶の容量を拡張することが可能である．

高い記録密度を持つ PCM であるが，PCM には書き込みアクセスに関する短所があり，単体で主記憶を構成することは困難である．PCM は DRAM に比べて書き込みアクセス時のレイテンシが大きい．また，素子の書き込み回数に制限がある．この 2 つの短所により，主記憶の構成素子として DRAM を単体で置き換えることは困難である．

この短所を隠蔽する方法として，PCM と DRAM を組み合わせて主記憶を構成するハイブリッドメモリアーキテクチャが検討されている [10–26]．このアーキテクチャでは，更新の多いデータを DRAM にオフロードすることで NVM への書き込みアクセスを削減し，NVM の短所を隠蔽しつつ，その利点を最大限に利用することができる．この目的のために，ハイブリッドメモリアーキテクチャにおいては，更新の多いデータを DRAM に，更新の少ないデータを NVM に配置することが求められる．もっとも単純な配置決定の方法は，メモリ上のデータに対する更新の回数を計測して，

DRAM と NVM 間でデータ配置を移動することである．Dhiman [10] の研究や Zhang [27] の研究では，ページング方式のメモリ管理方式を前提として，ページ単位での書き込み回数に基づいて DRAM と NVM の間でページの配置を移動する手法を提案している．

しかしながら，ページへの書き込み回数に基づいて NVM と DRAM との間でページ配置を決定する方法では，NVM の書き込みに関する短所を十分に隠蔽することができない．この方式では，ページへの書き込みアクセス回数を計測し，NVM にマッピングされていて，かつ書き込みアクセスが集中するページを DRAM へ移動する．そのため，実際にメモリに書き込みが発生するまで適切なページ配置が決定することができない．NVM への書き込みの集中から移動までの間に NVM への大量の書き込みアクセスが発生する．また，ページの移動はメモリ間コピーによって行われるため，移動の頻度によっては大きなオーバーヘッドが発生する．この問題を解決するためには，データが生成されたタイミングで，そのデータの更新頻度を予測することが必要である．

1.1.2 Web アプリケーションプラットフォームの複雑化

Web アプリケーションはアプリケーションソフトウェアの提供形態の一つである．Web アプリケーションは Web サーバによって実行され，その結果は動的な Web ページとしてユーザへ届けられる．従来はチケットの予約やグループウェアといった典型的なオンラインサービスに利用されることが主であったが，現在では，ワードプロセッサ，スプレッドシートといったこれまでクライアントの計算機環境でスタンドアロンに動作するようなアプリケーションも Web アプリケーションとして実現されている．

Web アプリケーションの発展に伴い，それをホストする Web アプリケーションプラットフォームのメモリ管理はより複雑化している．その理由の 1 つは，プラットフォーム管理者によるソフトウェアの挙動の把握が困難になりつつことである．Web アプリケーションプラットフォームを適切に管理するためには，動作するソフトウェアについての知識が必要である．しかしながら，Web アプリケーションの発展に伴ってソフトウェアの規模は大きくなっており，プラットフォーム管理者がホストする Web アプリケーションの挙動を把握することが難しくなってきている．また，Platform as a Service (PaaS) のようにプラットフォーム管理者とサービスの管理者が分離しているような環境 [28] では，不具合のあるソフトウェアや，既知の脆弱性に対する対応がなされていないソフトウェアが動作したり，悪意のある利用者による意図的な妨害活動がなされることもある．

2 つめの理由は，仮想化環境や Web アプリケーションランタイムの集密度が高いことである．Web アプリケーションのホスト形態はさまざまであるが，仮想化技術を用いて，1 つの物理サーバで複数のサービスをホストすることは珍しくない．ホストするサービスの数が多いほど，プラットフォーム提供者は多くの利益を上げることができる．また一つの Web アプリケーションに関して，リクエストを並行処理するために，Web アプリケーションランタイムのプロセスやスレッドが複数動作するのは，今日では標準的な構成である．アプリケーションランタイムのインスタンス数を大きくすることで処理の並列度が増加し，Web アプリケーションのリクエスト処理能力を向上できる．このため，一般的にサーバプラットフォームではメモリオーバーコミットが行われる．メモリオーバーコミットと

は，実際に搭載されている物理メモリより多くのメモリを仮想メモリとしてプロセスへ割り当てることである [29]．あるシステムで動作するプロセスは，その割り当てた仮想メモリ空間を常に全て使うわけではない．そのためメモリオーバーコミットによって，多くのプロセスや仮想化環境が並行動作することが可能である．しかしながら，プロセスや仮想化インスタンスの集密度が高い状況では，一部のプロセスや仮想化環境が引き起こす異常なメモリ消費が，システム全体の不定化を引き起こす．

このような仮想化環境やランタイムが集密度が高い環境では，従来の直近の利用状況のみ基づいたメモリリソースの管理だけでは不十分である．前述したように Web アプリケーションプラットフォームでは，管理者が関知しない大量のメモリ消費が発生する可能性がある．メモリオーバーコミットされた状態のサーバプラットフォームでこのような大量のメモリ消費が発生すると，ホストされている複数のサービスの可用性に悪影響を与える．そこで，異常なメモリ消費を行うプロセスを早い段階で検出し，そのメモリリソース利用を制限したり，仮想化インスタンスを停止することが必要である．現在の直近の利用状況のみに基づいたメモリリソース管理では，実際にメモリリソースの大量消費が発生するまで，それを検出することができない．もちろん，仮想化インスタンスが利用可能なメモリ量に制限を設け，その制限値の合計がホスト環境のメモリ領域より小さくなるように集密度を調整すれば，このような問題は防ぐことができる．しかしながら，プラットフォームの経済性を考慮すると，それは非現実的な選択肢である．

1.2　解決すべき課題と本研究でのアプローチ

第 1.1 節で述べた 2 つの問題に共通するのは，直近のメモリ利用状況のみに基づいたメモリ管理のみではメモリシステムの安定性を損なったり，その性能を十分に発揮できないことである．第 1.1.1 項で議論した NVM と DRAM を組み合わせたハイブリッドメモリにおけるメモリ配置の決定手法については，これまでの手法がメモリページへの書き込みアクセスの回数のみに基づいて仮想メモリページの配置を決定していたことが問題であった．第 1.1.2 項で議論した PaaS プラットフォームにおけるメモリ管理では，仮想化インスタンスについて，それぞれのメモリ使用量にのみ基づいてメモリ管理を行うことが問題であった．

本研究ではこの問題の解決にプロセスのリソース利用のふるまいを利用する．問題を解決するには，第 1.1.1 節での NVM への大量の書き込みアクセスや，第 1.1.2 節での問題のあるメモリ消費が実際に発生する前に，その兆候を捉えて対処する必要がある．本研究では，単位時間あたりのメモリ使用量の増分や，データごとに異なるメモリへの書き込みアクセスの傾向といった，プロセスのリソース利用のふるまいをその兆候として利用する．これらの情報は，プロセスがこれまでにどのようにメモリリソースを利用してきたかの特徴を表す情報であり，また同時にこれからどのようにメモリリソースを利用するのか，その特徴を表す情報である．この情報を利用すれば，異常なメモリ消費や，多量の書き込みリクエストなど，メモリリソースに関するイベントを，それが実際に発生する前に検出することが可能になる．そのため，前述した問題が解決可能である．

本研究では，プロセスのリソース利用のふるまいがメモリリソースの管理に有用であることを示すために，二つのふるまいに基づいたメモリ管理機構を設計し，実装と評価を行う．一つは，ハイブ

リッドメモリアーキテクチャのための，書き込みアクセスのふるまいに基づいたデータ配置決定機構である．第 1.1.1 項で述べた問題を解決するためには，メモリにデータを配置する時点で，そのデータに対して更新が多く発生するのか否かを予測する必要がある．そこでプログラミング言語処理系のレベルで得られるデータの意味を利用してデータに対する更新の多寡を予測し，NVM と DRAM の間でデータ配置を決定する手法を構築する．もう一つは，プロセスのメモリ消費のふるまいに基づいたメモリリソース隔離である．第 1.1.2 項で述べた問題を解決するためには，異常なメモリ消費を早い段階で検出することが必要である．そこで，プロセスごとの単位時間あたりのメモリ使用量の増分を監視し，急激なメモリ消費を行うプロセスを早い段階で検出し，他の正常なプロセスから切り離してリソース制限を加える手法を構築する．この手法を用いることで，実際に大量のメモリ消費が発生する前に，異常なメモリ消費に対応することが可能である．

1.3 本研究の貢献

本研究の貢献は以下の通りである．

- これまでリソース管理に利用されてきた直近の利用状況に加えて，リソース利用の経過情報から導かれるリソース利用のふるまいがメモリ管理に有用であることを示す．
- データの意味とその書き込みアクセスの傾向を利用したハイブリッドメモリアークテクチャにおけるデータ配置手法を確立する．
- メモリリソースのふるまいに基づいた新たなリソース管理手法である，プロセスのリソース隔離を確立する．

1.4 本論文の構成

第 2 章では，計算機システムの運用管理で用いられてきた種々のふるまいと，本研究で着目するリソース利用のふるまいについて述べ，その違いを議論する．

第 3 章では，リソース利用のふるまいを利用した，NVM と DRAM のハイブリッドメモリアーキテクチャにおけるデータ配置手法について述べる．

第 4 章では，リソース利用のふるまいを利用した，新しいメモリリソース管理手法である，リソース隔離について述べる．

第 5 章では，第 4 章で述べるプロセスのリソース隔離を応用した，Web アプリケーションに対する DoS 攻撃を防御する手法について述べる．

第 6 章では，プロセスのリソース隔離を応用した，fork 爆弾攻撃を防御する手段について述べる．この章で述べる提案手法は，は第 4 章，第 5 章で注目したふるまいとは異なる，間接的なリソース消費のふるまいに着目したリソース隔離の応用である．

第 7 章では，関連研究について述べる．

第 8 章では，本論文をまとめる．

第 2 章

計算機システムにおけるふるまいの利用

本研究では，メモリアーキテクチャの革新や，Web アプリケーションプラットフォームの管理の複雑化を背景としたメモリリソース管理上の諸問題について，メモリリソース利用のふるまいを利用した解決を試みる．本章では，まずこれまでの計算機システムにおける，種々のふるまいの活用について述べ，次に本研究が着目するメモリリソース利用のふるまいを定義する．

計算機システムの運用管理に，関連要素のふるまいを用いることは，これまでの様々な領域で行われてきた．また，システムへの侵入検知やマルウェア検出などの領域では，ネットワークトラフィックやプログラムのふるまいなどに着目した問題の検出が行われている．以下ではそれらふるまいについて述べ，本研究が着目するメモリリソース利用のふるまいとの相違点を述べる．

2.1　計算機システムの運用管理におけるふるまいの利用

計算機システムの運用管理において，過去の情報に基づき将来のプログラムの挙動を予測する工夫はさまざまな局面で見られる．以下ではその代表的な例を述べる．

キャッシュ

最も代表的なものは，参照の局所性 [30] に基づいたキャッシュ機構である．一度アクセスされたデータや，アクセスされたデータの近接データは，直近にアクセスされる可能性が高いため，より高速なアクセス手段にそのコピーを保持する．このようなキャッシュ機構は，プロセッサや，仮想記憶，ネットワークアプリケーションまで広く存在している．このキャッシュ機構は，参照の局所性というデータアクセスに関する一般的なふるまいに基づいた，プログラム挙動の予測を利用している．

ページ置換アルゴリズム

ページング方式の仮想記憶におけるページ置換アルゴリズムにも，プログラムのメモリアクセスのふるまいが利用されている．新たなページが要求され主記憶に空きページがない場合，オペレーティングシステムのメモリ管理機構は主記憶上のページを 2 次記憶に書き出し，空きページを確保する (ページアウト)．このページアウトの対象となるページを選択するアルゴリズムがページ置換アルゴ

リズムである．先述したキャッシュの例では，アクセスする可能性の高いデータを発見することを目的としているが，ページ置換アルゴリズムは反対にアクセスされる可能性の低いページを発見することを目的としている．

理想的には，全てのページについて次に使われるまでの時間を調べることで書き出しに最適なページを選択することができるが，現実にはその情報は得ることはできない [31]．そこで，スワップアウトの対象ページの選択には実際のメモリページへのアクセスのふるまいに基づいた予測を用いたアルゴリズムが利用されている．

分岐予測

プロセッサにおける分岐予測もふるまいを利用した例である．命令レベルの並列性を向上させるために命令パイプラインを実装したプロセッサは，パイプラインの効率向上を目的として，分岐予測に基づいた投機的機構を実装するのが一般的である．命令パイプラインを実装したプロセッサでは，命令フェッチ，デコード，実行，などに段階分けされた実行ユニットで，複数の命令を並列処理する．逐次的な命令実行の際はプログラムの順番に沿って命令を処理すればよいが，分岐命令の場合は，2通りの実行経路が生じる．分岐予測による投機的実行を実装したプロセッサでは，この分岐命令の実行にあたって，その方向を予測し，その予測した分岐先の命令を先行して実行する．

この分岐予測については，経験則に基づいた静的な方法や，状態機械を利用した動的な方法 [32] などがあり，これもプログラムのふるまいを利用した挙動の予測の一例である．

2.2　不正アクセスやマルウェア検出におけるふるまいの利用

計算機システムには，しばしば悪意のある第三者により不正アクセスが試みられる．システムへの不正アクセスとは，以下に示すような不正なシステム利用や，動作の妨害を指す．

- 本来ならば利用権限のない者が，直接的，あるいはネットワークを介して間接的に，何らかの方法でシステムを不正利用すること
- 利用権限に制限を課された利用者が，管理者が想定していない方法で，制限を超えた操作を行うこと
- システムに意図的に過負荷をかける，システムの不具合を利用して暴走させるなどして，システムの正常なサービス提供を妨げること

このような不正アクセスを完全に防ぐことは現実には困難である．管理者は，不正アクセスを未然に防ぐために，システムを点検し，設定の不備やシステムの脆弱性といった不正アクセスのきっかけとなる要素を排除しようとする．また，インターネットのように，不特定多数に公開されたネットワークに接続されたシステムの場合は，ファイアウォールを設けて通信を制限することも行う．しかしながら，一般にシステム設定の完全性を担保することは難しいこと，公表されていないソフトウェア脆弱性を利用したゼロデイ攻撃の存在などから，外部からの不正アクセスを完全に防ぐことは難しい．仮に外部からの不正アクセスは完全に防げたとしても，攻撃者が利用権限を持ったユーザーのシ

ステムの利用情報を盗んだり，利用権限を持ったユーザーが不正アクセスを試みる可能性もある．

システム管理者はこのような不正アクセスを未然に防ぐ努力をすると共に，その発生を早期に検出し，その不正アクセスに対処する必要がある．その手段として，Intrusion Detection System（IDS）の研究がこれまで進められてきた．IDS は対象システムへのネットワーク通信や，そのシステムで動作するプログラムを監視し，不正アクセスの兆候を検出する機構である．IDS を適切に利用することで，システム管理者は不正アクセスを早期検出することができ，その被害を最小限に食い止めることができる．コンピュータウイルスやワームといった悪意のあるプログラム（マルウェア）も，不正アクセス同様に，システム動作の正常性を阻害するものである．これらを検出する手法も IDS と同様にこれまで研究されてきた．

IDS やマルウェア検出の実現手法は，その検出ポリシーによって，シグネチャ型 [33–36]，異常検出型 [37–51] に分類される．シグネチャ型はあらかじめ定義された，不正アクセスの兆候やマルウェアの特徴を検出した場合に，問題発生を検出する方式である．異常検出型はシステムが正常に動作している状況をモデル化し，その正常モデルからのシステム状態の逸脱を問題として検出する方式である．

2.2.1 ふるまいに着目した異常検出

異常検出型の IDS やマルウェア検出では，システム状態の監視のために，計算機システムにおけるさまざまなふるまいを利用する．以下では，そのふるまいの種類ごとに，先行研究を述べる．

ネットワーク通信のふるまい

ネットワーク通信のふるまいは，対象システムに対する通信量やその種類の傾向である．これは異常検出で利用される代表的なふるまいの一つである．DoS 攻撃の発生が疑われる対象システムの通常状態を超えた通信の発生や，執拗なログイン再試行などを検出することで，問題を早期に検出することが可能である．

ネットワーク通信のふるまいを監視する IDS の方式にはいくつかの先行事例がある [37–41]．これらの方式は防御対象システムへのネットワーク通信を監視し異常を検出する点では同じであるが，その検出手法がそれぞれ異なっている．Zheng らの方式 [37] はベクトル量子化，Ioannis らの方式 [38] は統計的手法，Kang らの方式 [39] はサポートベクタマシンによる分類，Casas らの方式 [40] はクラスタリングと外れ値検出，Geramiraz の方式 [41] はファジー理論を用いて検出を行う．

システムコールのふるまい

システムコールのふるまいは，対象システムで動作するプログラムが実行するシステムコールの種類や，実行順序の傾向である．あるソフトウェアに関して，その実行するシステムコールの種類や実行順序，回数には一定の傾向がある．その傾向を監視し，正常状態からの逸脱を検出することで，サーバープログラムの脆弱性を利用した任意のプログラムを実行する攻撃や，実行ファイルを改変するマルウェアの動作を検出することが可能である．システムコールのふるまいを監視する IDS やマルウェア検出機構はいくつの先行事例がある [42–47]．

ストレージのふるまい

Pennigton らの方式 [48] では，ストレージのふるまいを利用してシステムへの侵入を検出する．この方式は，防御対象のシステムが外部のストレージサーバに接続されている環境を前提している．一般的に，攻撃者は侵入したシステムにおいて，監査ログを改ざんし，その侵入の痕跡を隠そうとする．また，バックドアプログラムの設置なども行う．このような操作には，特定のファイルへのアクセスやが伴うため，そのふるまいを監視することでシステムへの侵入を検出することができる．

ユーザのふるまい

ユーザがシステムを操作する際のふるまいも異常検出に利用可能である．例えばシステムにログインする際のパスワードを入力するキー操作を監視し，あらかじめ設定されたものと比較することで，ユーザアカウントが，本来の所有者ではない第三者に窃用されていることを検出できる．このようにシステムとユーザのインタラクションに関するふるまいを利用した IDS が提案されている [49–51]．前述したキー操作のふるまいだけでなく，Web ページの閲覧傾向，アプリケーションの使い方（Pannel の方式 [49]），ユーザごとの CPU 利用の傾向，GUI におけるウィンドウの数の傾向（Li の方式 [51]）などのふるまいなども利用される．

2.3 本研究におけるメモリリソースのふるまい

本研究で着目するメモリリソース利用のふるまいとは，メモリリソース利用の経過情報から得られるリソース利用の傾向である．メモリリソース管理の指標としては，これまでメモリ利用量が主に利用されていた．ここで定義するふるまいは，単なるメモリ利用量ではなく，ワークロードごとに異なるメモリの使い方を区別するための，高次情報である．本研究ではその中でも，セマンティクスに基づいたデータ更新のふるまい，プロセスのメモリ消費のスピードに着目する．

第 2.2 節，第 2.1 節で述べたとおり，計算機システムの運用管理にはさまざまなふるまいが活用されてきた．しかしながら，メモリリソース利用のふるまいを利用したものは，まだ存在していない．本研究ではそのメモリリソース利用のふるまいを活用して，第 1.1 節で述べた問題を解決する．

2.3.1 セマンティクスの書き込みアクセスのふるまい

計算機システムで実行されるプログラムでは，さまざまなデータが処理される．それぞれのデータは異なる意味を持ち，データが表現する内容（データのセマンティクス）ごとに，その更新の傾向は異なる．例えば，プログラムの実行時に与えられるパラメータを保持するデータは，実行中を通して更新が発生する可能性は極めて低い．その一方で，反復計算のためのデータには更新が多発することが見込まれる．このように，データの持つ意味によって，その更新の頻度を予測することが可能である．本研究では，このデータのセマンティクスごとに異なる更新の頻度を，セマンティクスの書き込みアクセスのふるまい，と定義する．

セマンティクスの書き込みアクセスのふるまいに基づいたデータ更新傾向の予測は， 第 1.1.1 項で

述べた DRAM と NVM によるハイブリッドメモリアーキテクチャの管理に有用である．ハイブリッドメモリアーキテクチャにおいては，更新の頻度を考慮した異種メモリ間でのデータ配置が求められる．特に，更新の多いデータは DRAM に配置する必要がある．ハイブリッドメモリにおける，データの更新頻度を考慮したデータ配置を行う単純な方法は，更新の多いデータを検出した時点で検出したデータを DRAM に移動する方法である．しかしながら，この方法では，実際にデータの更新が頻発した後でないと，問題を検出できない問題がある．この配置の決定にセマンティクスに基づいたデータ更新傾向の予測を利用することで，実際にデータに対する更新が発生する前に，データの更新が頻発する可能性のあるデータを検出することが可能である．

2.3.2　プロセスのメモリ消費のふるまい

プログラムが使用するメモリの量はそれぞれ異なり，また使用量の変化の傾向もそれぞれ異なる．そのメモリ利用の特徴を単位時間あたりのメモリ消費の変化で表したものが，メモリ消費のふるまいである．

メモリ消費のふるまいは，プログラムの異常動作の検出に有効である．プログラムは，異常動作により大量のメモリを急速に消費し，システムをメモリ枯渇状態に追い込むことがある．メモリ枯渇はシステムの不安定化を招き，場合によってはシステム全体が事実上の停止状態になることもある．特に第 1.1.2 項で述べたような独立したサービスが高い密度で管理されている場合，問題を起こしたサービスだけでなく，関係のないサービスまでもがその影響を受ける．システムの不安定化や停止を防止するためには，異常なメモリ消費を早い段階で検出する必要があるが，これはメモリ使用量のみに基づいたメモリ管理手法では困難である．プログラムのメモリ消費スピードを監視することで，この異常なメモリを検出することが可能である．

2.4　まとめ

過去の経過情報（ふるまい）に基づいて未来を予測することは，これまでの計算機環境の運用管理に広く用いられてきた．また，不正アクセスやマルウェアの検出にも用いられている．本章では，ふるまいを利用する先行手法を整理し，本研究で着目するふるまいについて定義を与えた．

第 3 章，第 4 章では，本章で定義したセマンティクスの書き込みアクセスのふるまい，プロセスのメモリ消費のふるまいを，それぞれ第 1.1.1 項，第 1.1.2 項で示した問題に適用して，解決することを試みる．

第 3 章

ハイブリッドメモリにおける セマンティクスに基づいたデータ配置

ふるまいに基づいたリソース管理の例の一つとして, 本章ではハイブリッドメモリアーキテクチャにおけるデータ配置について議論する. 第 1.1.1 項で述べた通り，NVM と DRAM のハイブリッドメモリアーキテクチャにおけるデータ配置には解決すべき問題がある．そこで本章では，データの意味（セマンティクス）を利用することで，データ配置時の書き込みアクセス予測の問題を解決することを試みる．プロセスに割り当てられたメモリ領域には，それぞれのプロセスが実行するプログラムのコンテキストでのセマンティクスを持ったデータを配置する．例えば，何らかのデータ構造，数値，文字列など，さまざまなものが考えられる．このセマンティクスには，書き込みアクセスの偏りがあると予想される．例えば，リンクリストの先頭ポインタを表すデータは頻繁に書き換えられる可能性がある一方，コマンドライン引数を表す文字列データは読み込みの対象となっても，書き換えの対象とはならない．このようなセマンティクスに基づいた書き込みアクセスの偏りを，プロセスごとのメモリアクセスのふるまいの 1 つとして見なし，ハイブリッドメモリにおけるデータ配置の決定に用いることで，効率的なデータ配置が可能になる．

この着想に基づき，データのセマンティクスを利用したハイブリッドメモリにおけるメモリ管理手法を設計，実装した．この提案手法はプログラムごとに固有のデータのセマンティクスを得るために，世代別ごみ集めを実装したオブジェクト指向プログラミング言語のランタイム拡張として設計した．ここでオブジェクト指向プログラミング言語を前提としたのは，オブジェクト指向言語にはクラス単位でのデータのセマンティクスがあり，より細かいデータのセマンティクスが得られるためである．提案手法が実装されたプログラミング言語ランタイムでは，実行するプログラムの命令を処理しながら，そのプログラムがメモリに足して行う書き込みアクセスを監視し，セマンティクスごとの書き込み傾向を把握する．世代別ごみ集めの実装を前提としたのは，世代別ごみ集めのメモリ管理の枠組みを応用することで，ハイブリッドメモリを効率よく管理可能であるためである．

本章の目的は，データのセマンティクスごとに異なる書き込みアクセスのふるまいが，ハイブリッドメモリにおけるデータ配置に有用であることを示すことにある．ハイブリッドメモリにおけるデータ配置処理を，言語処理系で実施するのか，オペレーティングシステム (OS) レベルで実施するのか，

その優劣を議論することは目的としていない．したがって本章では既存の OS レベルの方法と直接の比較は取り扱わない．本研究でオブジェクト指向言語処理系を用いたメモリ配置を議論しているのは，データのセマンティクスを認識してメモリ配置を決定できる手段であるためである．現在広く用いられている OS レベルのメモリ管理機構ではデータのセマンティクスを区別することができない．提案手法と OS レベルの方法を比較するには，より広い範囲のメモリ管理を扱う事ができるように，提案手法を拡張する必要があるが，本研究の段階では，提案手法を実装したプログラミング言語処理系で動作するプログラムのみを対象とする．

3.1 データのセマンティクスと書き込みアクセスのふるまい

データのセマンティクスはハイブリッドメモリにおけるデータ配置の決定に有用である．また，そのセマンティクスに基づいたデータ配置を決定するには，オブジェクト指向言語の言語処理系が有用である．以下では，データのセマンティクスが有効である理由や，言語処理系に着目する理由について述べる．

3.1.1 ハイブリッドメモリにおけるデータ配置が満たすべき要件

NVM と DRAM によるハイブリッドメモリの目的は，NVM の持つ書き込みアクセスに関する短所を隠蔽することである．そのため，ハイブリッドメモリを有効に利用するためには，NVM に対する書き込みアクセスを抑制すること，できるだけ多くのデータを NVM に配置すること，という 2 つの要件を満たしたデータ配置の管理が必要である．この 2 つの要件を満たすには，できるだけ多くのデータを NVM に配置しつつ，一方で書き込みの多いデータは DRAM に配置する必要がある

これまで，メインメモリにおけるデータ配置を司るのは主に OS の仕事であった．現在，広く用いられているページング方式による仮想メモリ機構では，プロセスに固定長のメモリブロックである仮想ページを割り当て，そのページの配置を OS が決定している．このメモリ管理の枠組みを利用して，ハイブリッドメモリにおけるデータ配置を決定する手法がこれまで研究されてきた．それらの手法はページ単位での書き込み傾向を監視し，書き込みの多いページを DRAM に移動する．しかしながら，その OS レベルの方法では，書き込みの多いページは，実際に書き込みが発生してからしか把握できない問題がある．つまり，もし書き込みの多いページが NVM に配置されていた場合は，それが OS に検出されるまで，ある程度の書き込みアクセスの集中が発生する．

3.1.2 データのセマンティクスを利用した，書き込みアクセスのふるまい予測

何らかの手段で書き込みの多いデータを予測できれば，書き込みが発生する前に適切な領域に配置することができる．そのデータの書き込み多寡の予測には，データのセマンティクスが有効である．データのセマンティクスとは，プログラムがそれぞれ利用するデータが持つ意味である．データのセマンティクスには二種類ある．一つはプログラミング言語レベルでのセマンティクスである．これは，データ型，クラスなど，プログラミング言語レベルで規定されるデータの意味である．もう一つ

の分類は，個別のプログラムレベルでのセマンティクスである．これは，それぞれのプログラムごとに異なる，データの意味である．二種類のセマンティクスはそれぞれ粒度が異なり，また包含関係にある．例えば，プログラミング言語レベルでは数値型であるデータ群は，プログラムレベルになるとそれぞれ異なる意味を持っている．

データのセマンティクスは，書き込みアクセスの多寡と相関があると考えられるデータの意味によっては，その用途からほとんど更新が発生しないと推定できるものや，反対に頻繁に更新されると推定できる．例として，何らかの反復計算をするプログラムを考える．このプログラムは，ユーザからパラメータを文字列で受け取り，それに基づいて，メモリ上のデータに対して反復計算を行う．このプログラムにおいては，文字列を表すデータは更新が少なく，反対に計算に用いる数値データに対しては，書き込みが多発すると予測できる．このように，データのセマンティクスに着目することで，前述した書き込み多寡の予測ができる．この予測に基づいてハイブリッドメモリに置けるデータ配置を行うことで，ハイブリッドメモリにおけるデータ配置を最適化することが可能である．

3.2 オブジェクト指向言語処理系を利用したセマンティクスの把握とデータ配置

第 3.1.2 項で述べた通り，データのセマンティクスはハイブリッドメモリにおけるデータ配置に有用である．しかしながら，に計算機システムにおけるメモリデータ配置決定を司る OS のレベルからはそのセマンティクスを得ることができない．一般的な OS はページの内容を認識しない．そのため，より高いレベル得られるセマンティクスの情報をカーネルに伝達するか，より高いレベルのメモリ管理機構にデータ配置の役割を委譲する必要がある．OS によっては，プログラムからデータの性質を伝える API を持つものがある．たとえば Linux では madvise() システムコールにより，プロセスからメモリページの利用傾向に関する情報を与えることができる．OS はその情報に基づいて，データ配置を決定することも可能であるが，しかしながら，プログラムレベルで明示的に指定する必要があり，汎用性は低い．

一方で，プログラミング言語処理系は，一般にデータ型の形でプログラミング言語レベルでのデータのセマンティクスを把握することができる．また，その中でも，オブジェクト指向言語の処理系は，クラスタイプという形でより詳細なセマンティクスを得ることができる．このセマンティクスと，書き込みアクセスのふるまいの関係性を利用することで，個々のデータに関する書き込みアクセス傾向の予測が可能になる．つまり，ハイブリッドメモリメモリにおける効率的なデータ配置が可能になる．

3.3 予備実験：データのセマンティクスと書き込みアクセス

本章での提案手法は, 以下の 2 つの仮説が正しいことを前提としている．

- データの書き込みアクセスのふるまいはそれぞれ異なり，偏りがあること
- データのセマンティクスごとのふるまいはそれぞれ異なり，偏りがあること

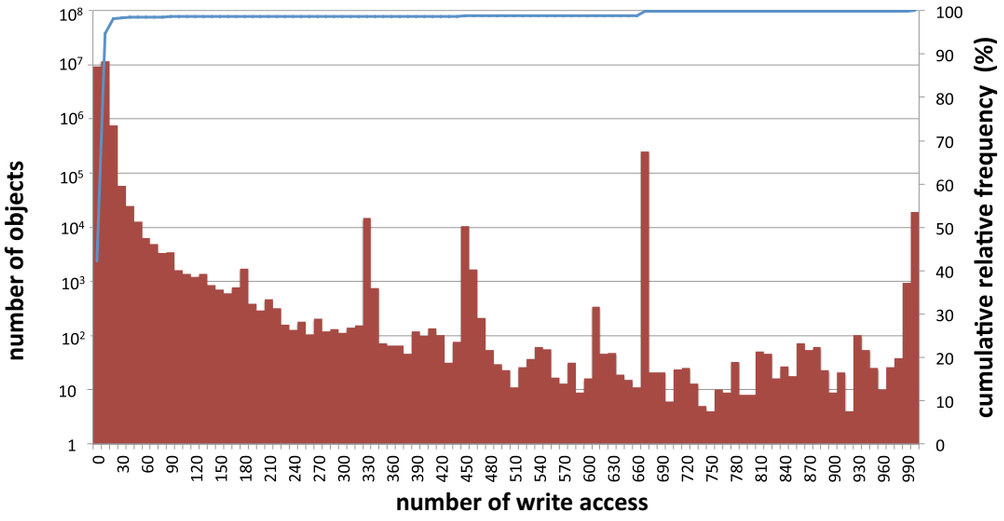

図 3.1　データごとの書き込み回数

この仮説が正しいことを検証するために, データごと, またセマンティクスごとの書き込みアクセスの傾向を調査する予備実験を行った.

予備実験は現実のプログラミング言語処理系の実装を変更することで行った. 利用した処理系は Ruby の処理系である,Rubinius 2.0.0. rc1 である. このプログラミング言語処理系の実装を変更し, 実験プログラムを実行した際のデータごと, セマンティクスごとの書き込みアクセスの回数を記録した. 実験プログラムとしては、Ruby Benchmark Suite を用いた. このプログラムは Ruby の処理系の性能を評価するためのベンチ—マークスイートである. このプログラムには Ruby の言語処理系が取り扱う典型的な処理が多く含まれており, 実際のプログラムを動作させたときのメモリアクセスと同じ傾向を示すことが期待できる. なおセマンティクスは, 言語処理系でのクラスで区別した.

データごとの書き込み回数を計測した結果を図 3.1 に示す. このグラフは対象ワークロードを実行した際のデータごとの書き込み回数をヒストグラムで表現したものである. 横軸は回数による階級である. 棒グラフと左縦軸で表現されるのが度数である. 例えば横軸での 0 の位置の縦軸の値は, データ配置後の書き込みアクセスが 0 回であったオブジェクトの数を示している. 青線と右縦軸で表現されるのは累積相対度数である. グラフからはワークロードの実行を通して生成されたオブジェクトのうち 98% は, 書き込み回数が 30 回以下であったことがわかる.

セマンティクスごとの書き込みアクセスを計測した結果を図 3.2 に示す. このグラフは対象ワークロードを実行した際のセマンティクスごとの書き込み回数の平均値を表現したものである. この実験で用いた言語処理系には 68 種類の標準クラスがあるが, ここでは平均値が 1 以上であったクラスについてグラフに示した. この結果から, データのセマンティクスごとにも書き込みアクセスのふるまいの偏りがあることがわかった.

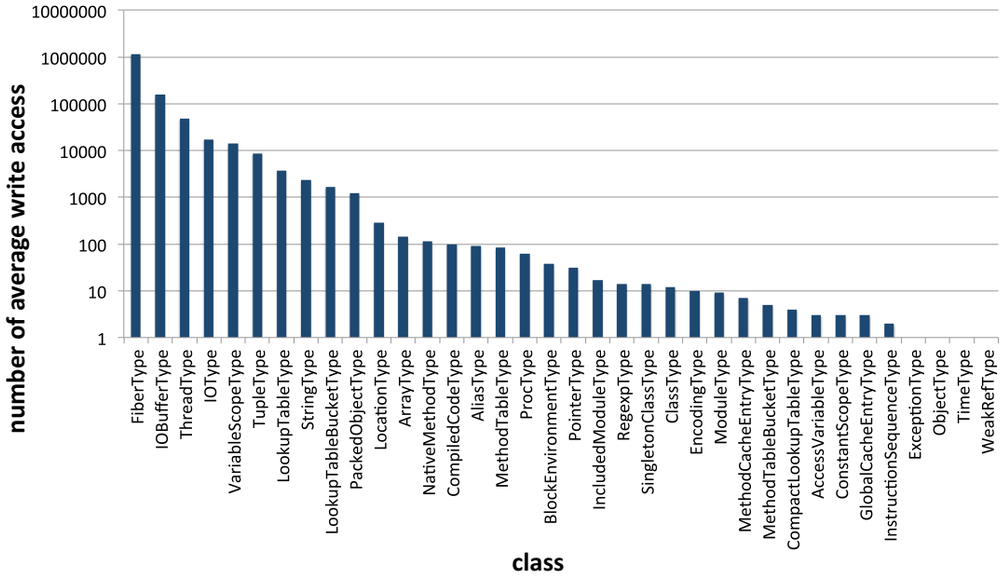

図 3.2　セマンティクスごとの書き込み回数

3.4　提案手法

予備実験の結果から, データのセマンティクスごとに書き込み傾向の偏りがあることがわかった．このセマンティクスごとの書き込み傾向を利用することで, データに対する書き込みアクセスのふるまいを, 実際に書き込みが発生する前に予測することが可能である．この結果に基づき, セマンティクスごとの書き込みアクセスのふるまいを利用して, データ配置を決定する手法を設計した．本節では，まず提案手法の概要を説明し，次に，提案手法を構成する機能の詳細について述べる．

3.4.1　提案手法の概要

提案手法は，コピーごみ集めとマーク・アンド・スイープ方式の組み合わせによる世代別ごみ集めを実装したプログラミング言語処理系を前提とする．世代別ごみ集めは，生成されたオブジェクトのほとんどはすぐに参照されなくなるが，ある程度参照が続いたオブジェクトはその後も参照され続ける可能性が高いという経験則 [52] に基づいたごみ集め戦略である．世代別ごみ集めでは，オブジェクトは新世代領域と旧世代領域に分けられたヒープの中に配置される．全てのオブジェクトは基本的にに新世代領域に生成され，しばらく有効であったオブジェクトは旧世代領域に移動される（オブジェクトの昇格）．新世代領域はメモリ利用効率が低いが，処理コストが小さいコピー GC で管理され，旧世代領域は処理コストが大きいが，メモリ利用効率が高いマーク・アンド・スイープ方式で管理される（図 3.3）．

提案手法では，世代別ごみ集めの機能をハイブリッドメモリの管理に応用する（図 3.4）．利用す

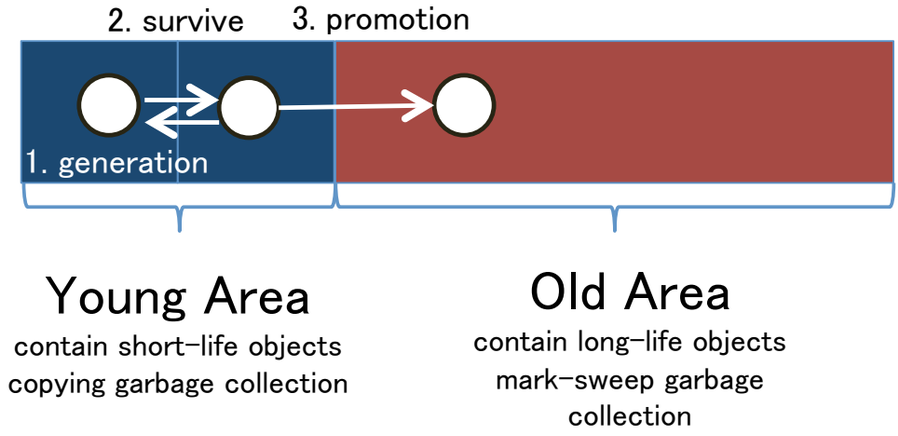

図 3.3　世代別ごみ集めの概要

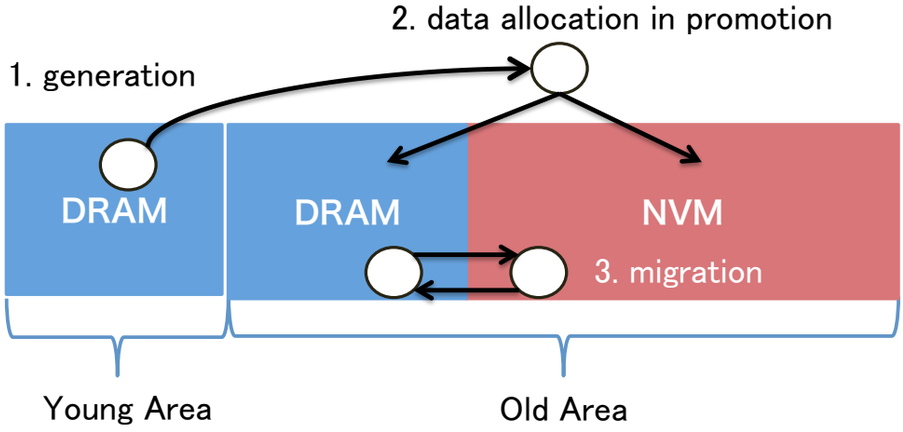

図 3.4　提案手法の概要

るのは，オブジェクトに対する書き込みアクセスを検出するライトバリア処理と，旧世代領域に対するマーク・アンド・スイープ方式によるごみ集め処理である．前者はオブジェクトに対する書き込み傾向を把握するために利用し，後者は DRAM と NVM の間でオブジェクトを移動するために利用する．

ハイブリッドメモリの管理における問題である，データ配置時の書き込みアクセスの多寡の予測は，世代別ごみ集めにおけるオブジェクトの昇格処理時に行う．全てのオブジェクトは新世代領域に生成され DRAM に配置される．このオブジェクトが旧世代領域に昇格するタイミングで，そのオブジェクトのクラスを調べ，もしそのクラスが書き込みが多いクラスであるならば DRAM へ，そうでない場合に NVM で配置する．書き込みが多いクラスは，後述するオブジェクトマイグレーション処理時に決定される．

旧世代領域に配置されたオブジェクトに対して書き込みアクセスが発生すると，言語処理系は，オ

ブジェクトの管理情報が持つ書き込み回数のカウンタを増加させる．また同時に，書き込みが発生したオブジェクトを特殊なキュー構造に登録する．もしすでにキューに登録されていた場合には，登録位置が更新される．このデータ構造は，それぞれレベルを持つ階層状のキューで構成されており，書き込みが集中するオブジェクトほど上位のキューに，反対に書き込みが少ないオブジェクトについては下位のキューに集まるように設計されている．提案手法では，このキューを用いてオブジェクトの書き込み傾向を把握する．

NVM と DRAM に分割された旧世代領域では，実際の書き込みの多寡に基づいてオブジェクトの移動が実施される．オブジェクトの昇格時には，昇格対象のオブジェクトのクラスに対する書き込みの傾向に基づいてそのオブジェクトに対する更新の多寡を予測し，NVM と DRAM の間でデータ配置を決定する．しかしながら，この予測は外れる可能性がある．また，プログラムの実行初期段階では，どのクラスに書き込みアクセスが集中するのかの情報がない．そのため，全てのオブジェクトは一旦 NVM に配置される．その中には更新の多いオブジェクトも含まれる．このようなケースに対応するために，NVM に配置されていて更新が集中するオブジェクトは DRAM に移動される．オブジェクトが移動対象であるか否かは，前述した階層状のキューを用いて判定する．また，移動されたオブジェクトのクラスは書き込みが多いクラスであると判定され，同じクラスのオブジェクトはオブジェクト昇格時の領域選択で DRAM に配置すべきオブジェクトとして判定されるようになる．反対に，DRAM に配置されながら，実際には書き込みアクセスが発生しないオブジェクトもある．このようなオブジェクトは DRAM から NVM へ移動される．このオブジェクトの移動は旧世代領域に対するマーク・アンド・スイープごみ集め処理時に実施される．また，オブジェクトの実際の移動処理はマーク・アンド・スイープ処理の機能を利用して行う．

3.4.2　メモリの割り当て

提案手法が想定するメモリ割り当てを図 3.5 に示す．提案手法では新世代領域の GC にコピー GC を使用することを前提としている．このコピー GC は領域を 2 つの空間に区切り，片方を From，もう片方を To 空間とする．新しく生成されたオブジェクトは From 空間に配置され，GC 時には有効なオブジェクトのみを To 空間にコピーする．有効なオブジェクトを全てコピーが完了すると，From と To の役割は交代し，GC 前に From であった空間のオブジェクトは全て無効になる．新世代領域ではこのコピー GC が頻繁に実行され，データのコピーに伴うメモリへの書き込みが頻繁に起こると推定される．そのため，新世代領域は NVM への配置に適さない．提案手法では，新世代領域のオブジェクトはオブジェクトごとの書き込み傾向に関わらず全て DRAM に配置されるものとする．一方，旧世代領域についてはマークアンドスイープ方式で管理され，オブジェクトの移動は起こらない．そこで旧世代領域については，DRAM に割り当てる空間と NVM に割り当てる空間に分割し，オブジェクトへの書き込み傾向に基づいて配置を決定する．

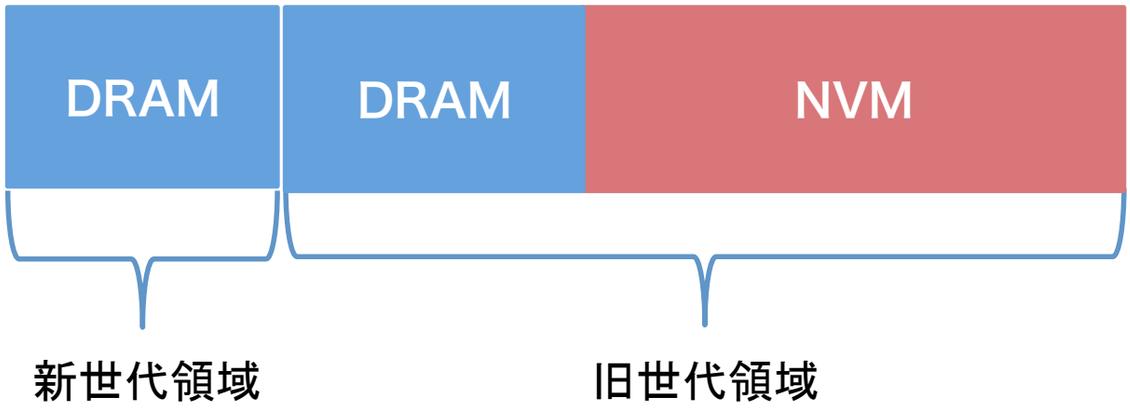

図 3.5　提案手法でのメモリ割り当て

3.4.3　オブジェクト昇格時の配置決定

第 3.4.2 項で述べたように，提案手法では旧世代領域のオブジェクトを NVM，DRAM のいずれかの領域に配置する．この配置の決定は，オブジェクトが新世代領域から旧世代領域に移動する際に行われる．この配置の決定は，移動対象のオブジェクトのクラスが，DRAM 配置クラスであるか否かを判断することで行われる．

DRAM 配置クラスとは，書き込みアクセスが集中すると推定されるクラス群である．DRAM 配置クラスは最初は空である．後述する NVM から DRAM へのオブジェクトマイグレーションが発生した際に，その移動対象となったオブジェクトが所属するクラスが，この DRAM 配置クラスに登録される．

3.4.4　NVM における書き込み傾向の収集と記録

第 3.4.3 項で述べたように，提案手法では新世代領域のオブジェクトが旧世代領域に移動する際に NVM，DRAM 間での配置を決定する．しかしながら，この移動時の配置決定だけでは NVM への書き込みを抑制するという目的を達成できない．その理由は，所属するクラスの書き込み傾向から外れるオブジェクトが存在するためである．また，プログラムによっては，配置のタイミングでは書き込みの少ないクラスであっても，その後の進行によって書き込みが多いクラスに変化する可能性もある．このような場合に対応するために，後述する NVM，DRAM 間でのオブジェクトの移動を行う．

このオブジェクトの移動の対象となるオブジェクトを検出するために，NVM に配置したオブジェクトの書き込み傾向を収集する．書き込みを検知する手段としては，世代別ごみ集めのライトバリア処理を応用する．ライトバリア処理は，オブジェクトに対する書き込みが起こる度に呼び出される．この処理時に，オブジェクトごとの書き込み回数を計測する処理を追加する．

追加する処理は，書き込み回数の記録と，write-hot queue の更新である．書き込み回数の記録

は，書き込みが発生したオブジェクトの管理情報が持つ，書き込み回数のカウンタを増加させる．write-hot queue は旧世代領域に配置されたオブジェクトの書き込みアクセスの傾向を把握するための，特殊なデータ構造である．write-hot queue についての詳細は後述する．

3.4.5 write-hot queue

write-hot queue は NVM に配置されたオブジェクトの書き込み傾向を把握するためのデータ構造である．このデータ構造は先行研究で提案された，多段キューによる書き込み傾向の把握で利用されているデータ構造である [14, 53].

図 3.6 は write-hot queue の概要を示したものである．このデータ構造は，m 個のキューで構成されており，それぞれのキューは n 個のエントリをそれぞれ持っている．n はキューの深さであり，MQ_DEPTH と定義する．m はキューの段数であり，MQ_NUM と定義する．それぞれのキューは相異なったレベルを持っている．最も低いレベルのキューは Q_0 と表記し，最上位のキューは Q_{max} と表記する．Q_0 と Q_{max} の間に位置するキューは，Q_1，Q_2... のように，Q_0 から Q_{max} の方向へ数を数えて表記する．

不揮発メモリに配置されたオブジェクトに対する書き込みアクセスを検出したとき，そのオブジェクトへの参照が Q_0 の先頭に挿入される．もし，そのオブジェクトへの参照がすでにキューに存在していた場合，その参照は Q_i に移動する．提案手法では $i = \log_2 N$ とする．ここでの N は，対象オブジェクトへの書き込みアクセス回数である．この i の定義は先行研究 [14, 53] にて使用されたものである．もし，オブジェクトへの参照が最も高いキュー (Q_{max}) に存在していた場合は，その参照を Q_{max} の先頭に移動する．

write-hot queue では，定期的にキューの降格が行われる（図 3.7）．降格が起こると，キューに登録されいるオブジェクトへの参照は，1 つ下のレベルに移動される．キューの降格の目的は，時間経過を考慮に入れるためである．例えば，あるオブジェクトへの参照を Q_p から $Q_{p\ -1}$ に降格させるとする．この時，最下位の Q_0 は取り除かれる．また，移動した際に，書き込み回数のカウンタを半分に減らす．これも時間経過を考慮するために行う．

第 3.4.4 項で述べた，旧世代領域に配置されたオブジェクトに対する書き込みの検出時に，この write-hot queue を更新することで，書き込みの頻度が高いオブジェクトへの参照は write-hot queue の上位に，反対に書き込みの少ないオブジェクトへの参照は write-hot queue の下位に集まる．

3.4.6 配置後のオブジェクトマイグレーション

第 3.4.4 項で述べたように，旧世代領域に配置されたオブジェクトの中には，その書き込みアクセスの傾向によって，配置後に DRAM と NVM 間で移動されるものがある．この操作をオブジェクトマイグレーションと呼ぶ．

オブジェクトマイグレーションには二つの方向がある．一つは，NVM から DRAM へのオブジェクトの移動である．この移動では，NVM に配置されていながら，書き込みが多いオブジェクトが移動される．書き込みが多いオブジェクトは，第 3.4.5 項で述べた write-hot queue を用いて検出する．

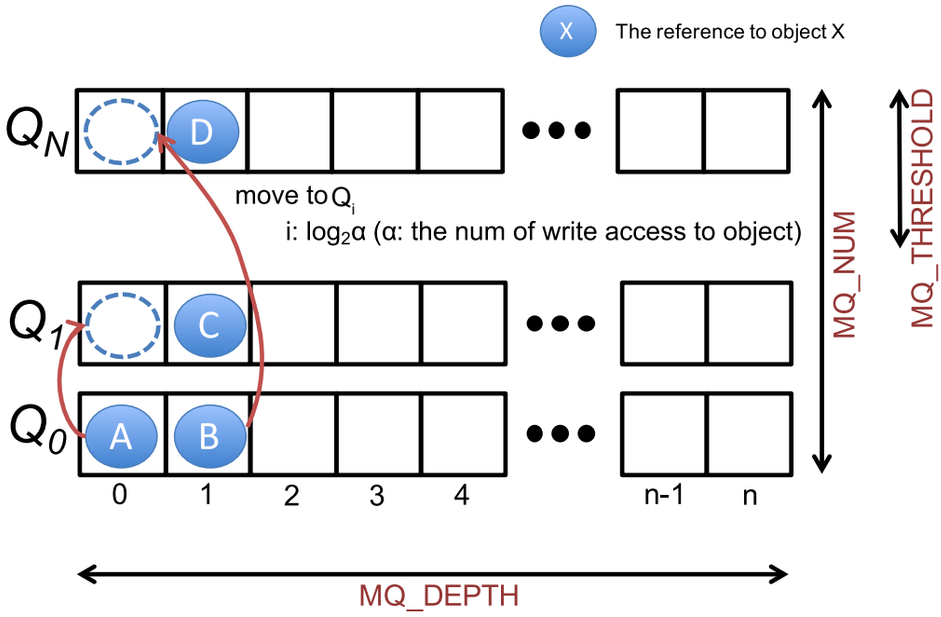

図 3.6　write-hot queue

write-hot queue のうち，上位 n 個のキューに登録されている NVM 配置オブジェクトを書き込みが多いオブジェクトであると見なし，移動の対象とする．このパラメータ n を MQ_THRESHOLD と定義する（図 3.6）．また，このときマイグレーションの対象となったオブジェクトが所属しているクラスは，書き込みの多いクラスとして 3.4.3 項で述べた DRAM 配置クラスに登録される．

もう一方のオブジェクトマイグレーションは，DRAM から NVM へのオブジェクトの移動である．この移動では DRAM に配置されていながら，書き込みが少ないオブジェクトを移動する．前述した通り，NVM から DRAM へのマイグレーション対象は，ライトバリア処理を応用することで検出できるが，DRAM から NVM へのマイグレーション対象はライトバリアでは検出することができない．なぜなら，書き込みの少ないオブジェクトはライトバリア処理を起こさないためである．そのため，DRAM に配置されたオブジェクトを全て走査し，書き込みの少ないオブジェクトを検出する必要がある．一般にこのオブジェクトの走査はコストが大きいため，提案手法では，このオブジェクトの走査をごみ集め処理に便乗して実施する．

オブジェクトマイグレーションは，旧世代領域におけるごみ集め処理の拡張として実現する．旧世代領域ではマーク・アンド・スイープ方式のごみ集めが行われる．このごみ集め処理は，旧世代領域に存在する有効なオブジェクトを全走査して印を付ける処理（マーク処理）と，印が付かなかったオブジェクトを消去する処理（スイープ処理）で構成される．提案手法では，この全走査をオブジェクトの移動と DRAM での書き込みの少ないオブジェクトの検出に利用する．ごみ集めが開始されると，言語処理系は write-hot queue を調べ，NVM から DRAM へ移動されるべきオブジェクトの管理情報に，マイグレーション対象である印を記録する．その後，マーク処理時に，もしマイグレーション対象である印がついている，NVM に配置されたオブジェクトを発見した場合には，そのオブ

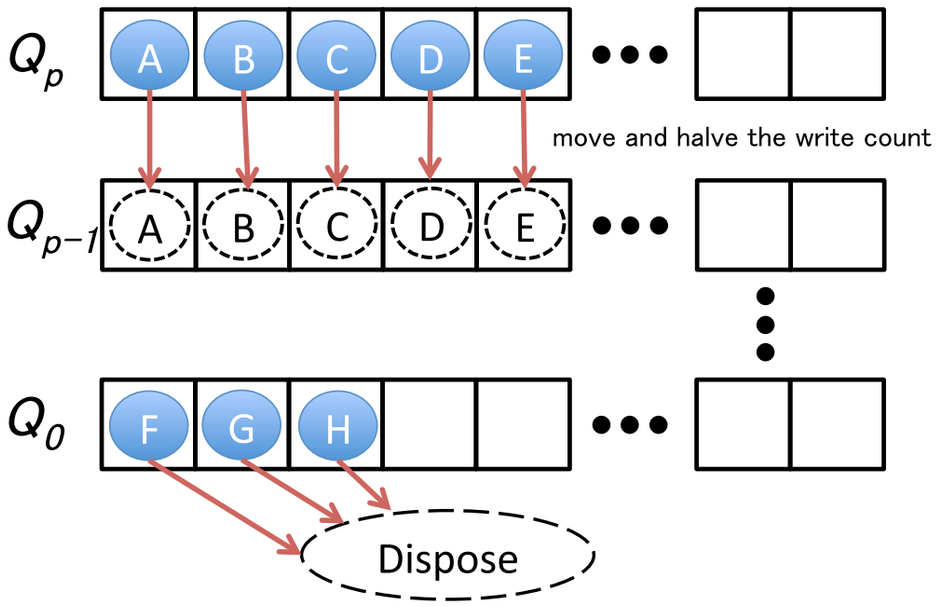

図 3.7　キューの降格

ジェクトを NVM から DRAM へ，また反対方向へ移動させる．また DRAM に配置されたオブジェクトを検出した場合には，その書き込みアクセスの回数を調べ，2 回以下の書き込みしか発生していない場合は，マイグレーション対象である印を付けて，NVM から DRAM へのマイグレーションと同様に，DRAM から NVM へマイグレーションを実行する．

3.4.7　旧世代領域におけるごみ集め実行機会の追加

第 3.4.6 項で述べたように，DRAM と NVM の間でのオブジェクトマイグレーションは，旧世代領域のごみ集め処理のタイミングで実施される．このごみ集め処理は，旧世代領域のメモリ使用量に基づいて実行される．そのため，NVM に対する書き込みアクセスが集中するような状況になったとしても，旧世代領域のメモリ使用量が基準より小さければ，オブジェクトマイグレーションは実施されない．これは NVM への書き込みアクセスを抑制する観点からは不適切な挙動である．また，ハイブリッドメモリの目的を果たすためには，DRAM の使用量は少ない方が望ましい．DRAM へのオブジェクト配置が集中した場合にもオブジェクトマイグレーションを実施し，NVM へオブジェクトを移動する必要がある．

適切な局面でオブジェクトマイグレーションを実施するために，提案手法ではごみ集めの実行条件を追加した．追加した条件は二つである．一つは，NVM へ書き込みが集中したときである．もう一つは，旧世代領域の DRAM へのオブジェクト配置が基準値を超えた場合である．前者は，write-hot queue の上位キューに含まれる NVM 配置オブジェクトの割合が DRAM 配置オブジェクトの割合を超過したときである．後者は言語処理系が管理するヒープ領域のうち，DRAM が占める割合が一定以上になったときである．

表 3.1　実験環境

CPU	Intel Core i7-4960X (6 core, 12 thread)
RAM	32.0GB
Swap	なし
OS	Linux 4.0.5 (Gentoo Linux)

3.5　評価実験 1

提案手法を既存の言語処理系に実装し，その効果を評価する実験を実施した．本実験の目的は，提案手法の実装した言語処理系を用いることにより，ハイブリッドメモリアーキテクチャにおける NVM への書き込みアクセスを抑制できるかどうかを検証することである．そのために，提案手法を既存の言語処理系に実装し，ベンチマークプログラムを動作させた際のメモリへの書き込みアクセスの状況や，メモリの利用量などを計測し，評価した．実装の対象には世代別ごみ集めを実装した Java の言語処理系である Jikes RVM [54] を用いた．実験は表 3.1 に示す計算機環境で実施した。

なお，現段階では主記憶として利用可能な NVM は一般に流通していないため，本実験では実際の NVM は使用していない．実験は DRAM 単体で主記憶を構成したシステムで実施し，言語処理系が管理するメモリ領域の一部分を NVM であるとみなしてデータ配置をシミュレーションする．NVM への書き込みは実際には DRAM に対して行われる．そのため，実験結果には NVM のアクセス特性は再現されていない．

3.5.1　対象ベンチマーク

実験は言語処理系の性能を評価する dacapo benchmark suite(dacapo) [55] を用いた．これは Java のプログラミング言語処理系の実行性能を計測するためのベンチマークセットとして広く用いられている．本実験ではバージョン番号 2006-10-MR2 を利用した．

dacapo は複数のワークロードで構成されている．それぞれのワークロードは独立したプログラムとして動作する．表 3.2 に dacapo に含まれるベンチマークの処理内容を示す．表に示したワークロードの他にも chart というワークロードがあるが，グラフィカルインタフェースを用いたワークロードであるため，実験環境の制約から，対象外とする．

実験では，それぞれのワークロードを独立して実行した．Jikes RVM はマルチコア環境でのスレッド実行に対応しているが，本実験ではそれぞれの試行で使用できるコア数は最大で 1 とした．これはベンチマークの実行に割り当てられるコア数が試行ごとに変化することを防止するためである．

表 3.2　dacapo benchmark suite のワークロード一覧

ワークロード	内容
antlr	ANTLR（構文解析器生成プログラム）によるデータ処理
bloat	Java バイトコード（コンパイル済みプログラム）の最適化
eclipse	JDT（統合開発環境 eclipse における Java 用サポートツール）に含まれるパフォーマンステスト
fop	Apache FOP を用いた XML による組版処理
hsqldb	銀行のトランザクション処理を模したデータベース処理
jython	Java で動作する python 処理系上でのベンチマーク（pybench）の実行
luindex	Apache Lucene（全文検索のためのクラスライブラリ）による文書の索引作成
lusearch	Apache Lucene を用いた文書検索処理
pmd	PMD（Java のソースコード解析器）を用いたソースコード解析
xalan	Apache Xalan を用いた XML 文書の HTML 変換

3.5.2　書き込みアクセスのふるまいの観察

第 3.3 節で述べた実験では，オブジェクト指向言語処理系におけるデータ配置の単位には書き込みアクセスのふるまいに関して偏りがあることを示した．また，オブジェクトが所属するクラスによっても書き込みアクセスのふるまいに偏りがあることも示した．ここでは提案手法を実装した処理系に対して同様の実験を実施し，評価実験の環境でも同様の書き込みアクセスのふるまいの偏りがあることを確認する．

実験は提案手法を実装した言語処理系上で実験対象のベンチマークを実行し，生成されたオブジェクトに対する書き込みアクセスの回数を記録することで実施した．この実験の際には，提案手法が実現する DRAM と NVM 間でのオブジェクト配置の決定は行わない．生成されたオブジェクトは新世代領域に生成され，昇格処理の際に旧世代領域へ移動される．このそれぞれの領域でプログラムの実行中に無効になったオブジェクトの書き込みアクセスの回数を収集した．また，プログラムが終了する時点での有効な全てのオブジェクトに関して書き込みアクセスの回数を収集した．

図 3.8 は，ベンチマーク項目 antlr に関して収集したオブジェクトに対する書き込みアクセス回数のヒストグラムを示したものである．青棒にはプログラムの実行中に生成されたオブジェクトの書き込み回数を 10 回単位で分類し，その頻度を左縦軸に示した．左縦軸は対数表示である．なお，1000 回以上の書き込みアクセスの頻度は，1 つにまとめた（グラフ右端）．緑線は青棒で示すデータに対する累積相対度数である．このグラフから，ベンチマーク antlr に関して 8 割を超えるオブジェクトについて書き込みアクセスの回数が 20 回以下であり，ほぼ全てのオブジェクトに対する書き込みアクセスの回数が 200 回以下であったことがわかる．一方で全体のうちの少数のオブジェクトに関しては書き込みアクセスの回数が多く，このようなオブジェクトを検出し，適切に配置することでハイブリッドメモリを効果的に利用可能である．

図 3.9 は，ベンチマーク項目 antlr に関して収集したオブジェクトに関する書き込みアクセス回数に基づいて，クラスごとの平均書き込みアクセス回数を計算し，そのヒストグラムを示したものである．青棒が示すデータは，収集したオブジェクトに対する書き込みアクセス回数のデータをクラスで分類，合計し，その合計をクラス毎のオブジェクト数で除したものである．緑線の意味は図 3.8 と同様である．このグラフから，提案手法が対象とする言語処理系においてもクラスごとに書き込みアクセスのふるまいに偏りがあることがわかる．全体のうち 8 割を超えるクラスが，平均書き込み回数が 30 回以下であるのに対して，平均して 1000 回以上の書き込みアクセスが発生したクラスも確認できた．このクラスごとの書き込みアクセスのふるまいの偏りを利用することで，セマンティクスごとに異なる書き込みアクセスのふるまいに基づいてデータに対する書き込みアクセスの多寡を予測することが可能である．

図 3.10〜3.26 は，同様の分析を全てのベンチマーク項目に実施した結果である．いずれのベンチマーク項目においてもオブジェクト単位での書き込みアクセスのふるまいが偏っていることが確認できる．また，クラスによっても書き込みアクセスのふるまいが異なることがわかる．

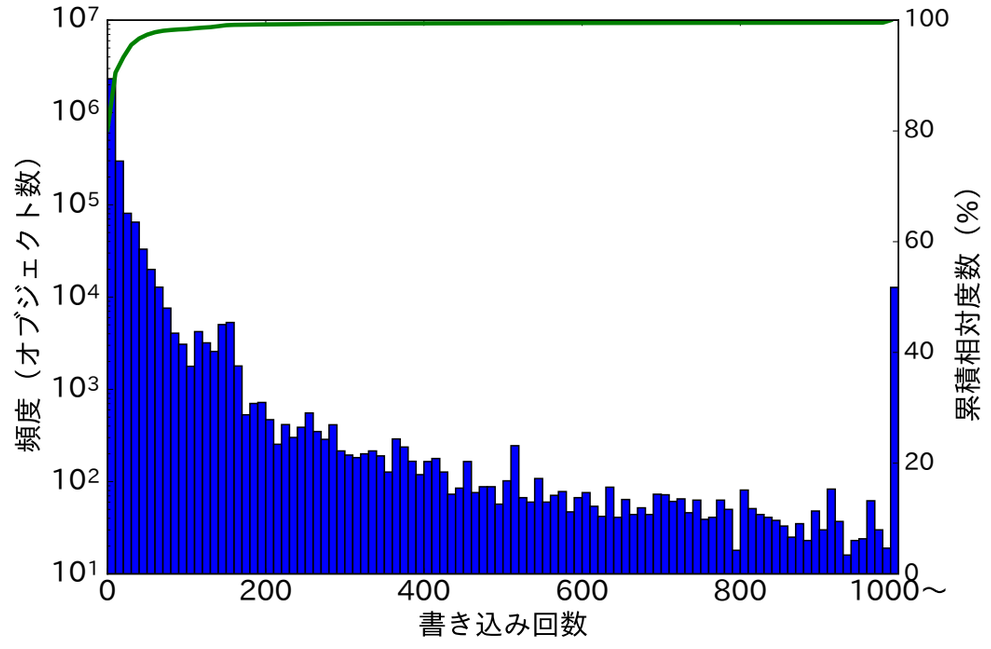

図 3.8　オブジェクト単位での書き込みアクセス回数の分類（antlr）

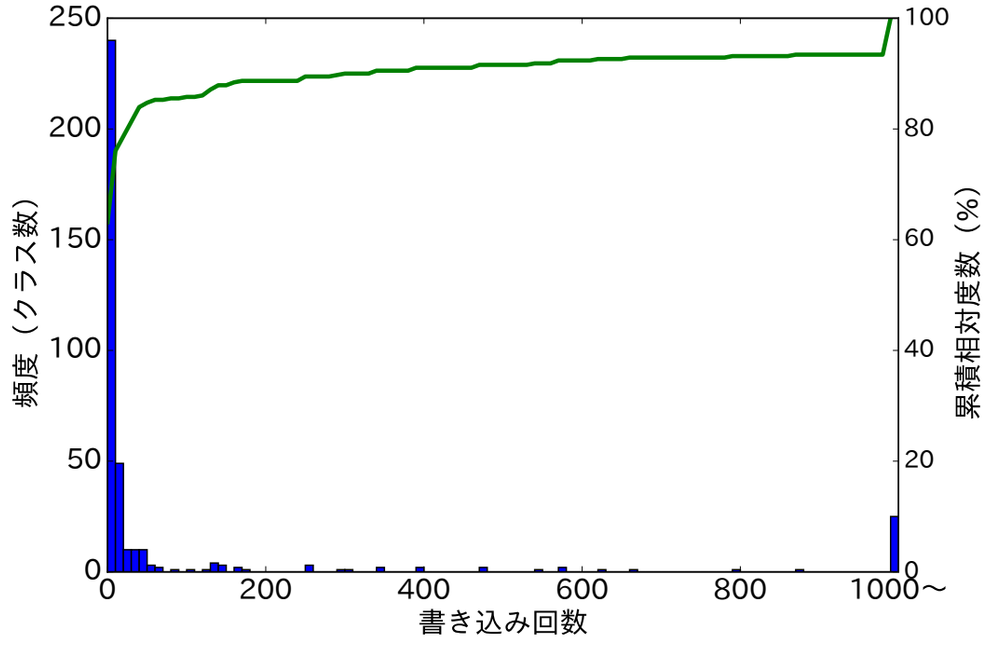

図 3.9　クラス単位での書き込みアクセス回数の分類（antlr）

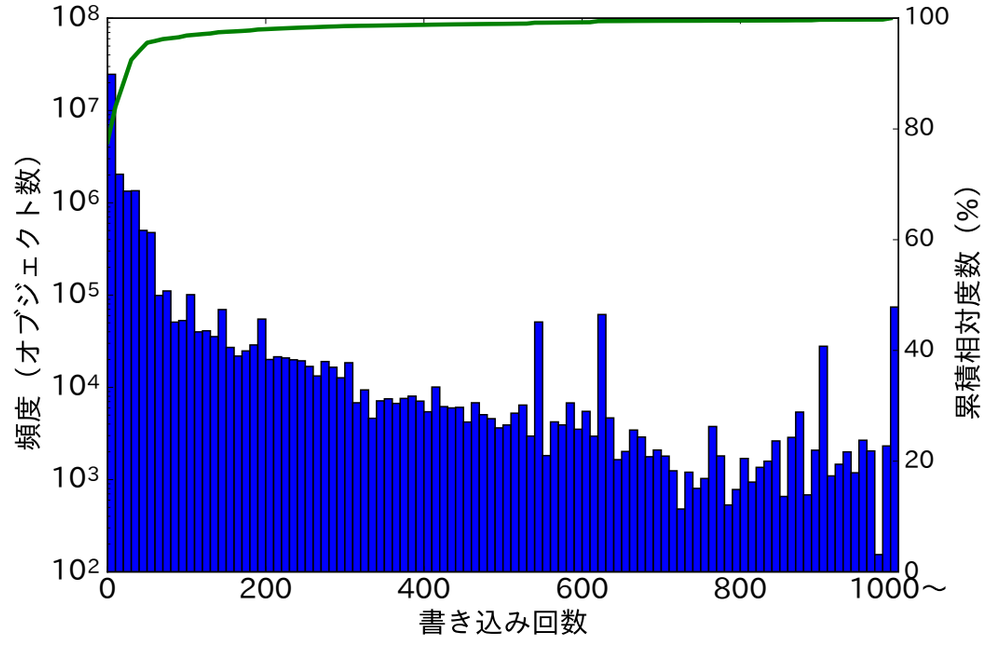

図 3.10　オブジェクト単位での書き込みアクセス回数の分類（bloat）

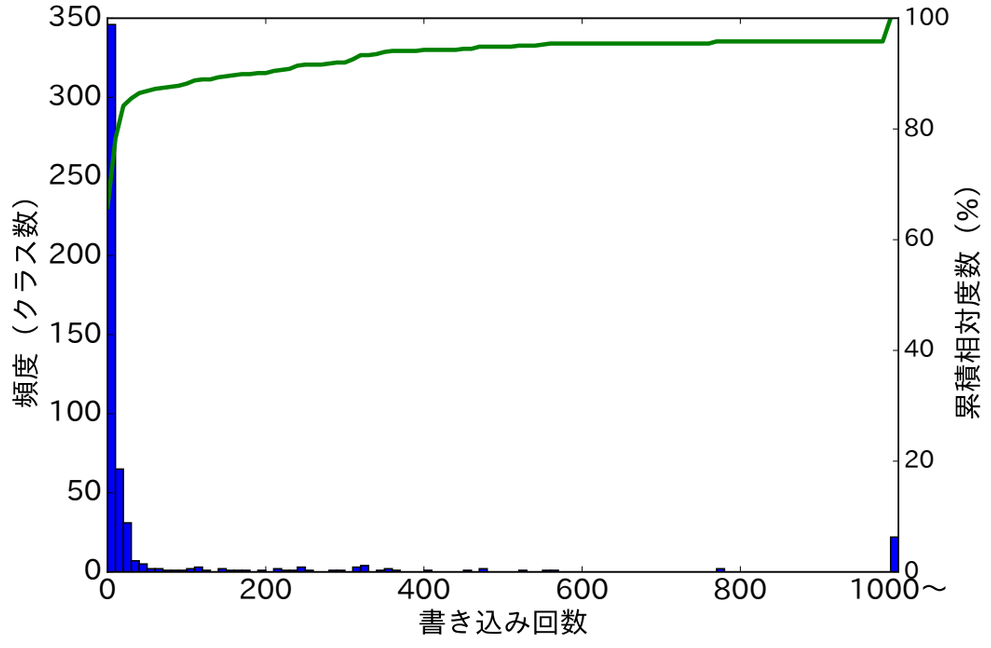

図 3.11　クラス単位での書き込みアクセス回数の分類（bloat）

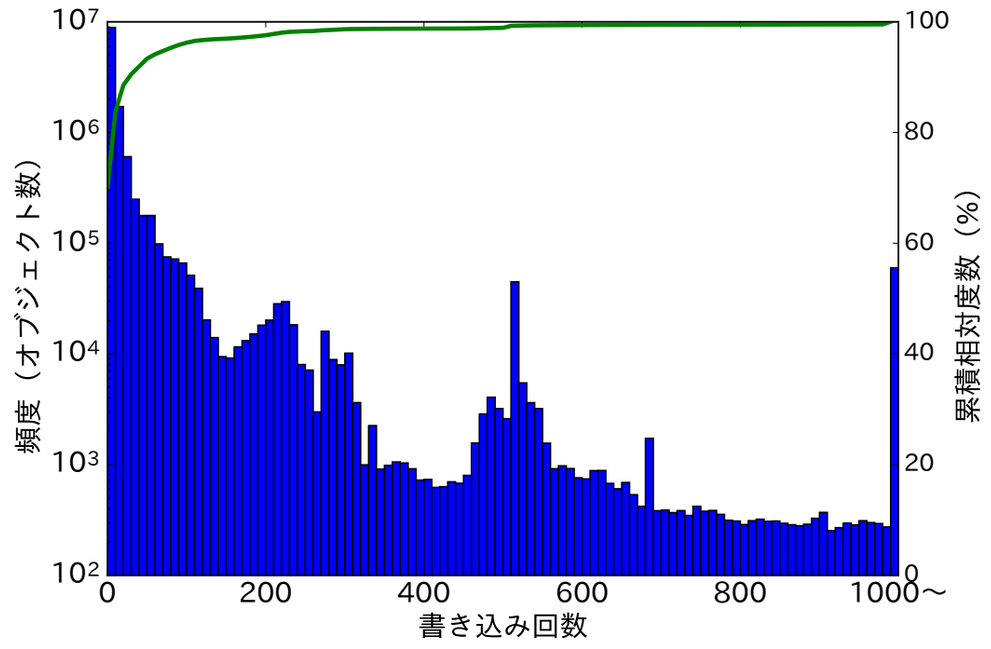

図 3.12　オブジェクト単位での書き込みアクセス回数の分類（eclipse）

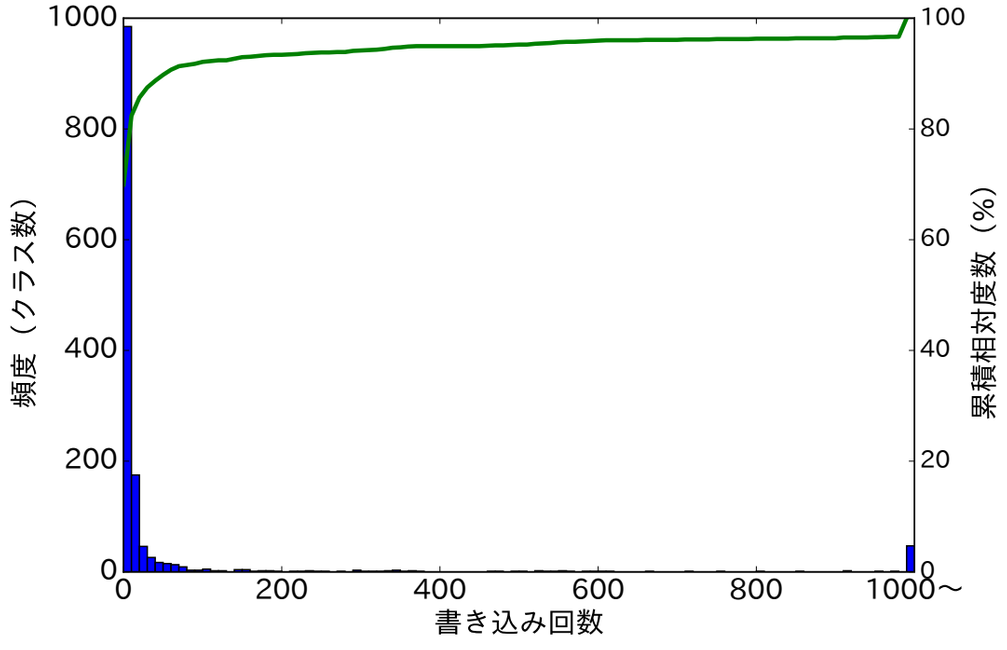

図 3.13　クラス単位での書き込みアクセス回数の分類（eclipse）

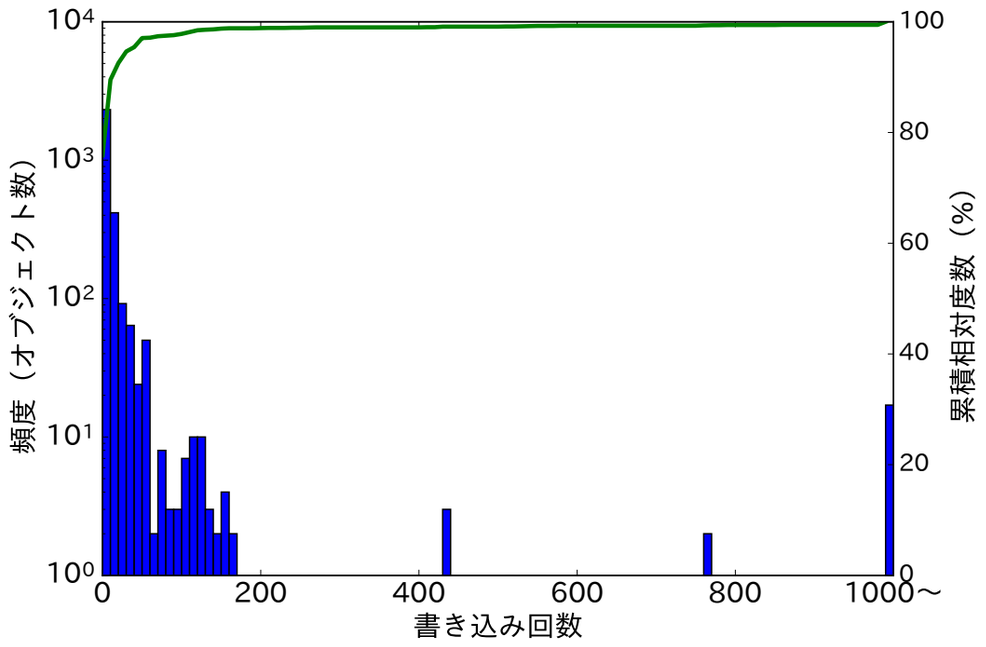

図 3.14　オブジェクト単位での書き込みアクセス回数の分類（fop）

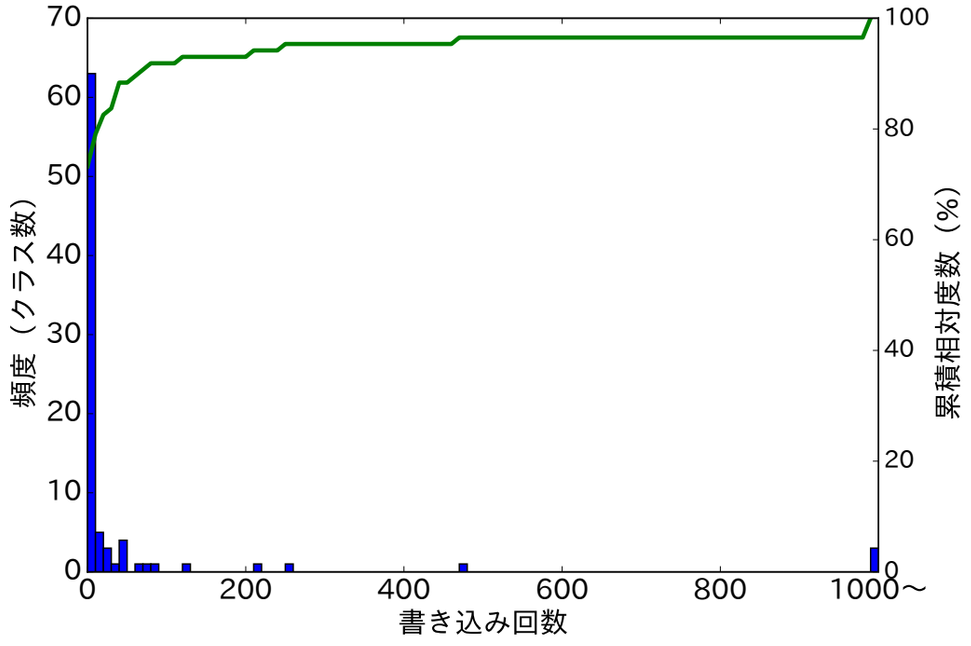

図 3.15　クラス単位での書き込みアクセス回数の分類（fop）

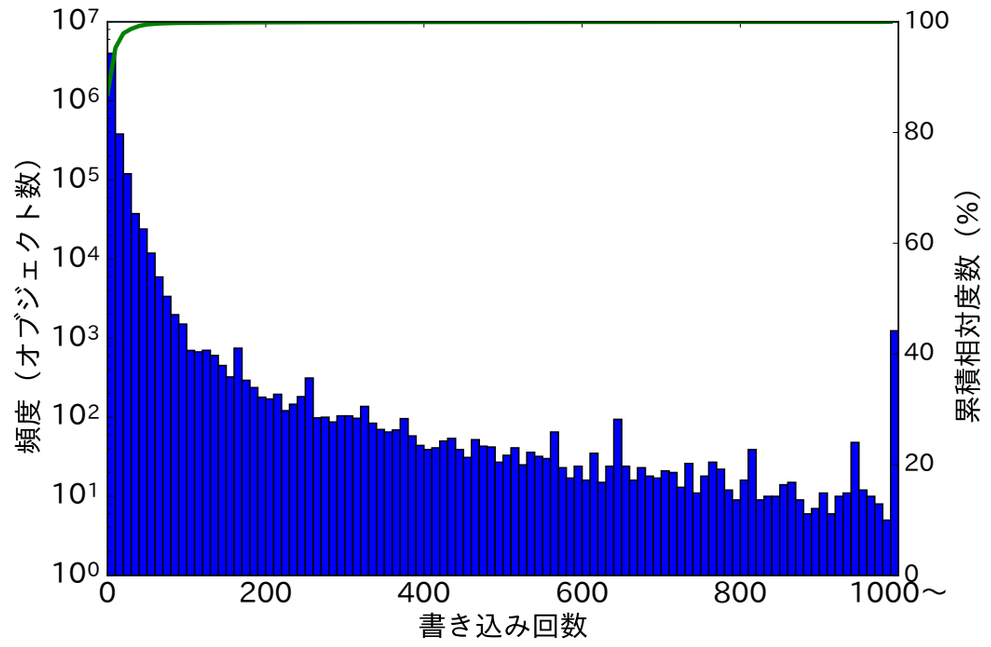

図 3.16　オブジェクト単位での書き込みアクセス回数の分類（hsqldb）

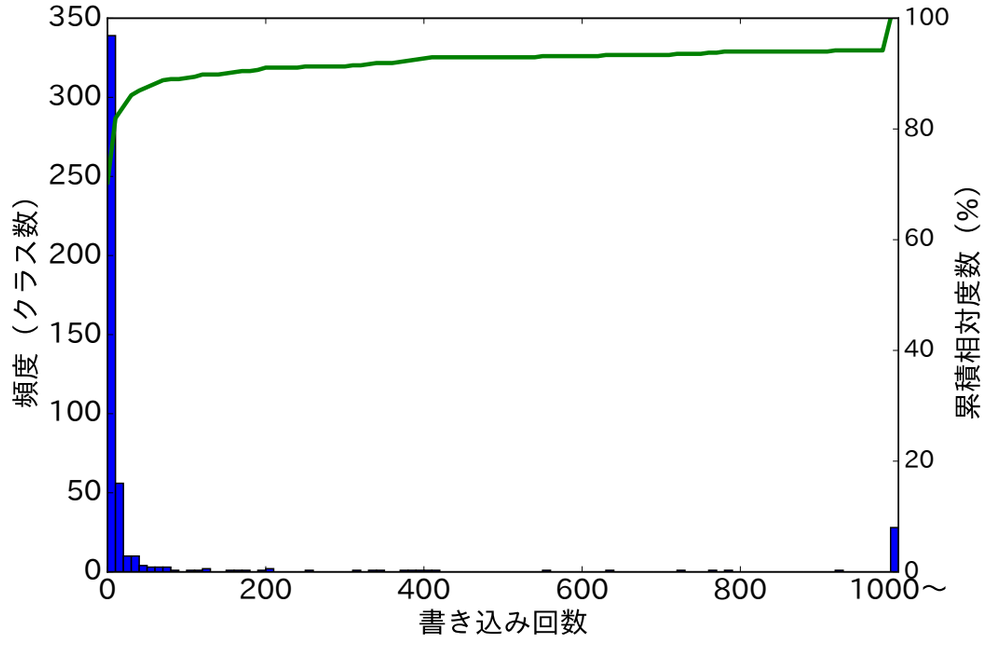

図 3.17　クラス単位での書き込みアクセス回数の分類（hsqldb）

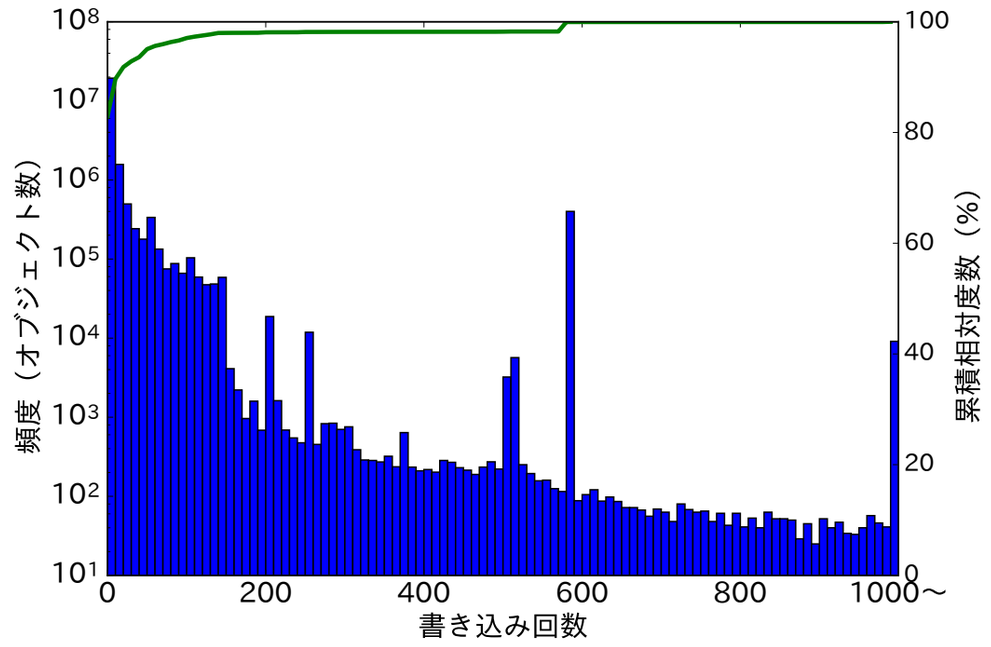

図 3.18　オブジェクト単位での書き込みアクセス回数の分類（jython）

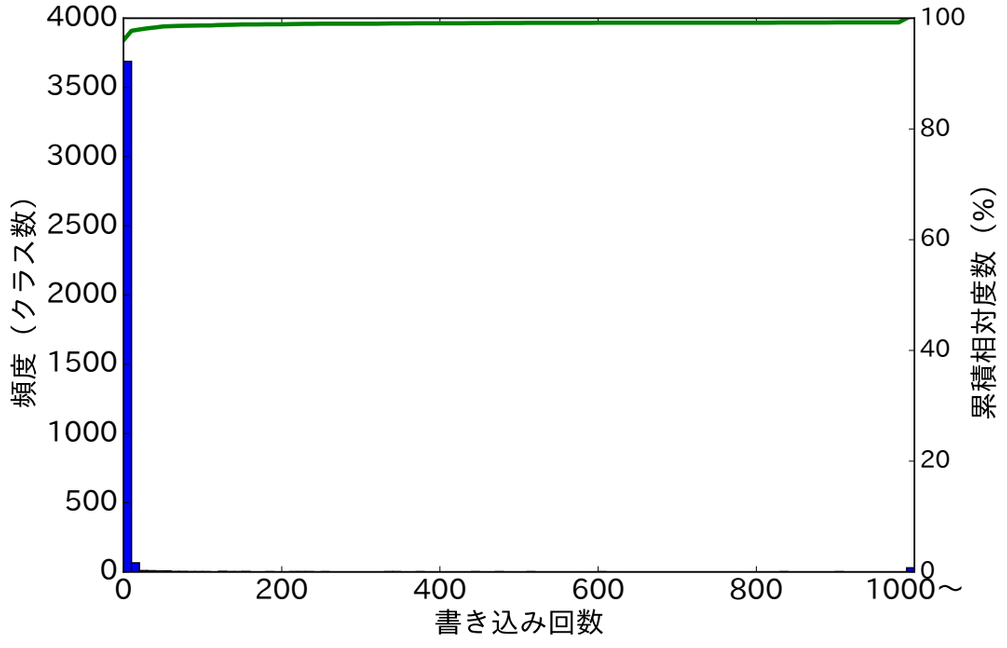

図 3.19　クラス単位での書き込みアクセス回数の分類（jython）

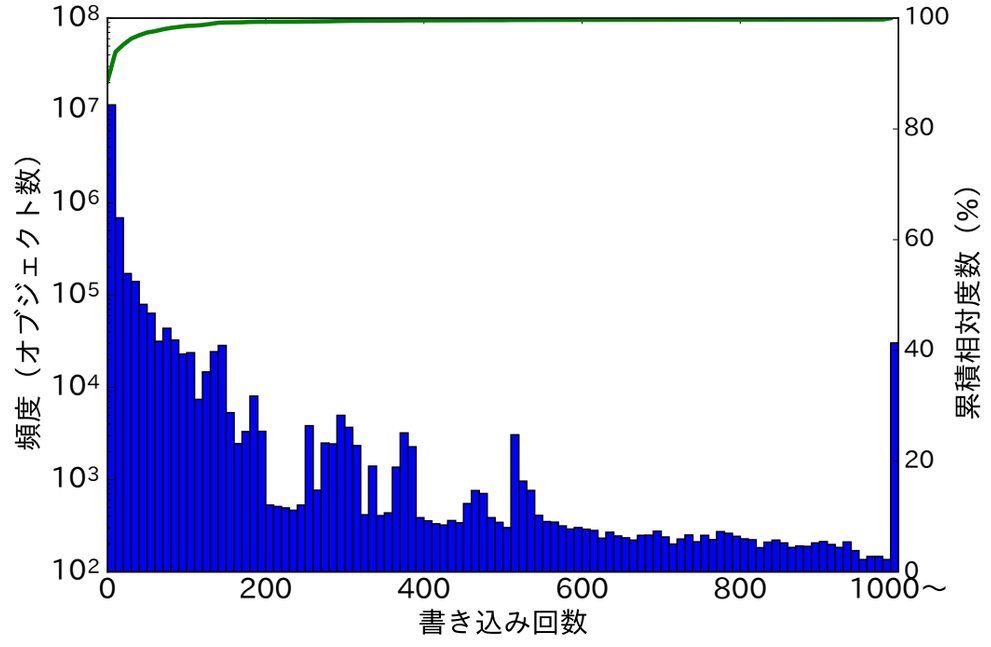

図 3.20　オブジェクト単位での書き込みアクセス回数の分類（luindex）

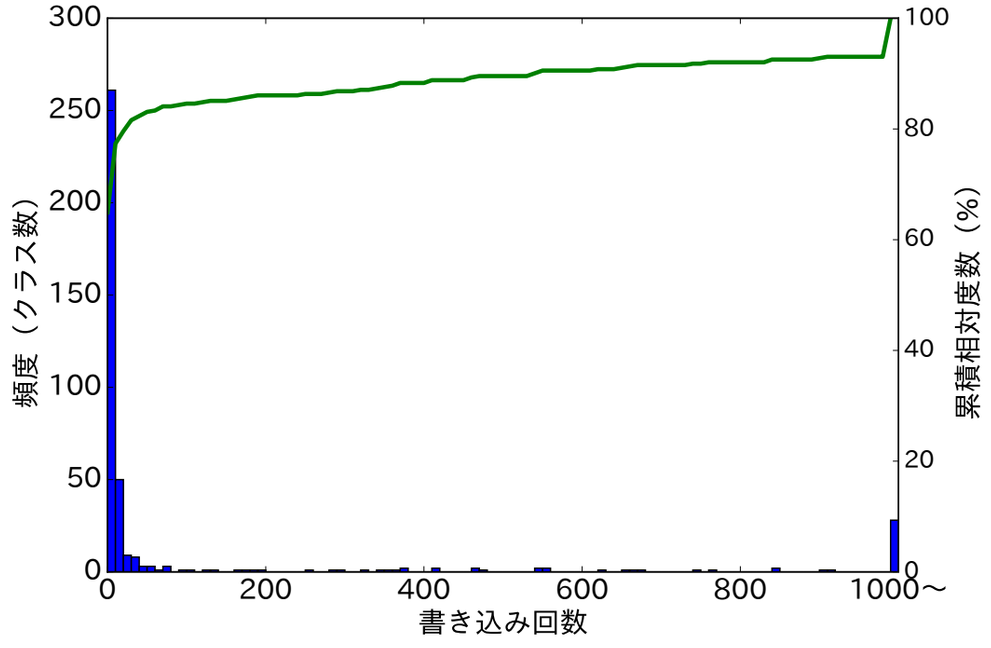

図 3.21　クラス単位での書き込みアクセス回数の分類（luindex）

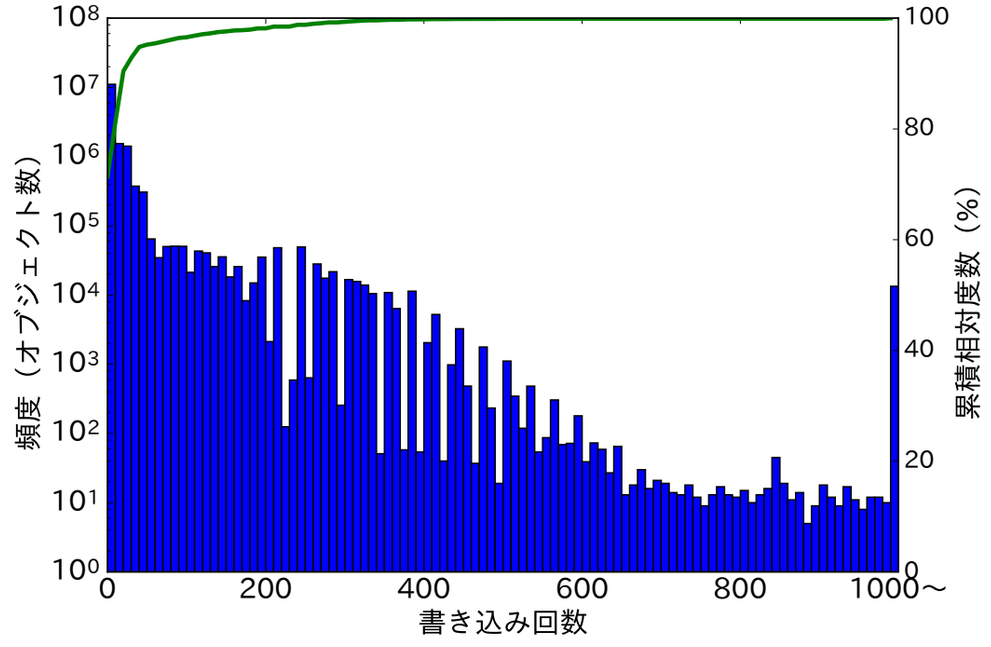

図 3.22　オブジェクト単位での書き込みアクセス回数の分類（lusearch）

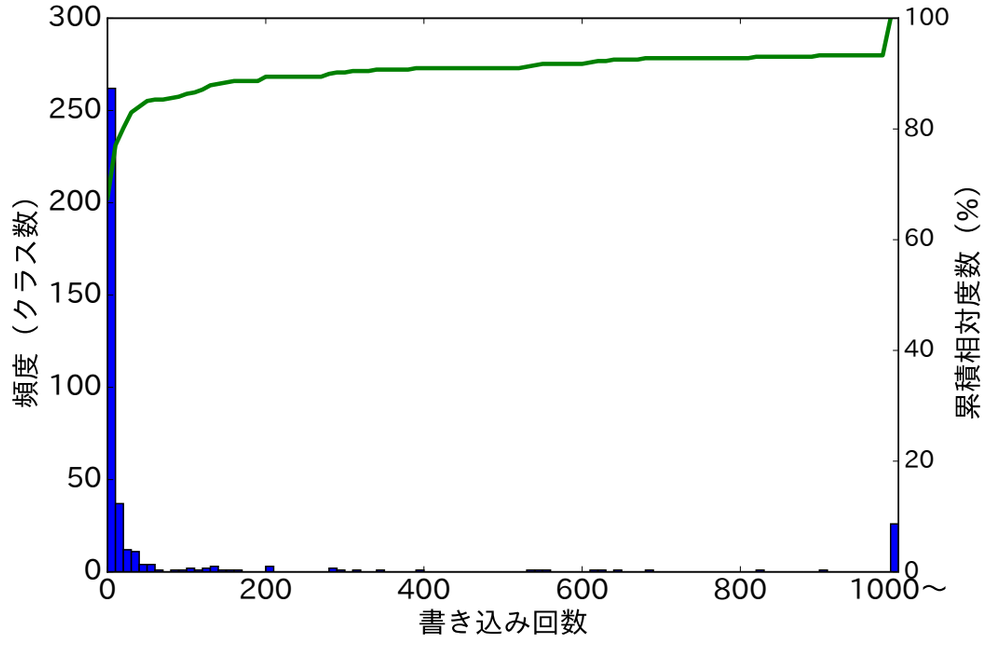

図 3.23　クラス単位での書き込みアクセス回数の分類（lusearch）

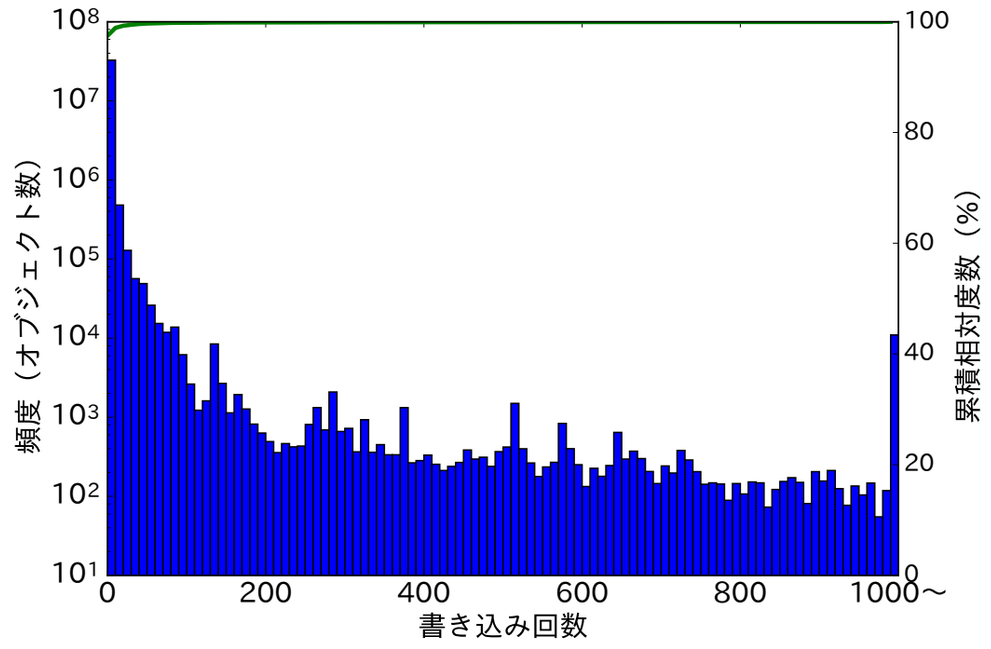

図 3.24　オブジェクト単位での書き込みアクセス回数の分類（pmd）

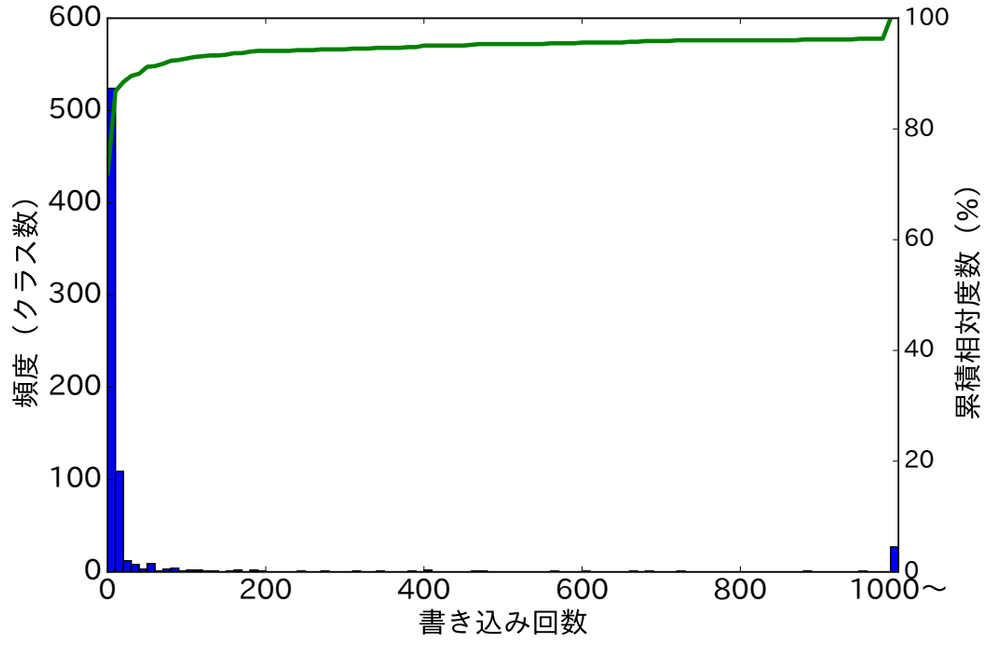

図 3.25　クラス単位での書き込みアクセス回数の分類（pmd）

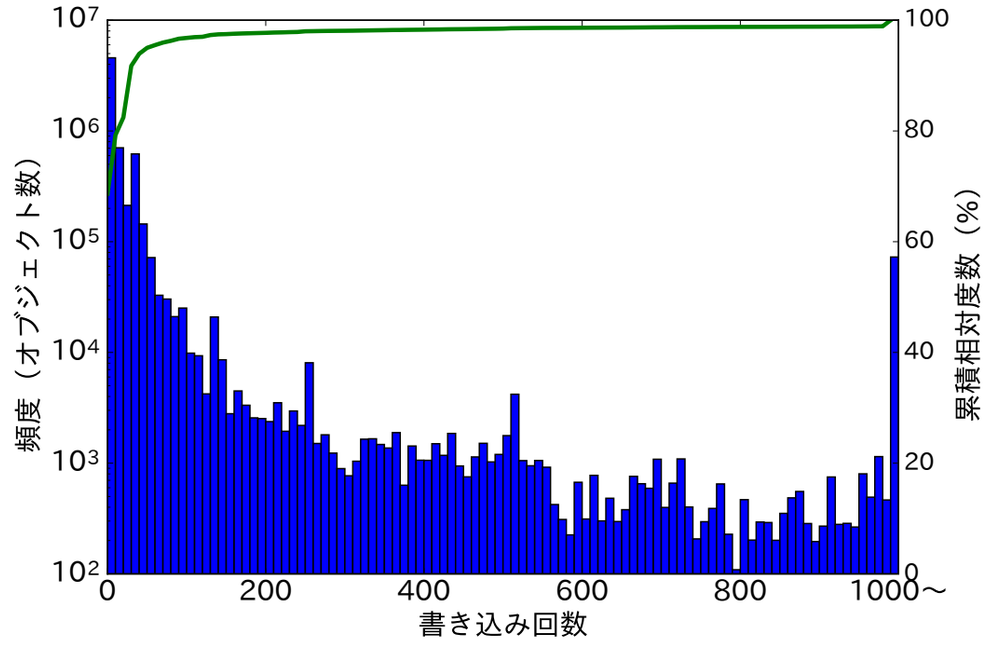

図 3.26　オブジェクト単位での書き込みアクセス回数の分類（xalan）

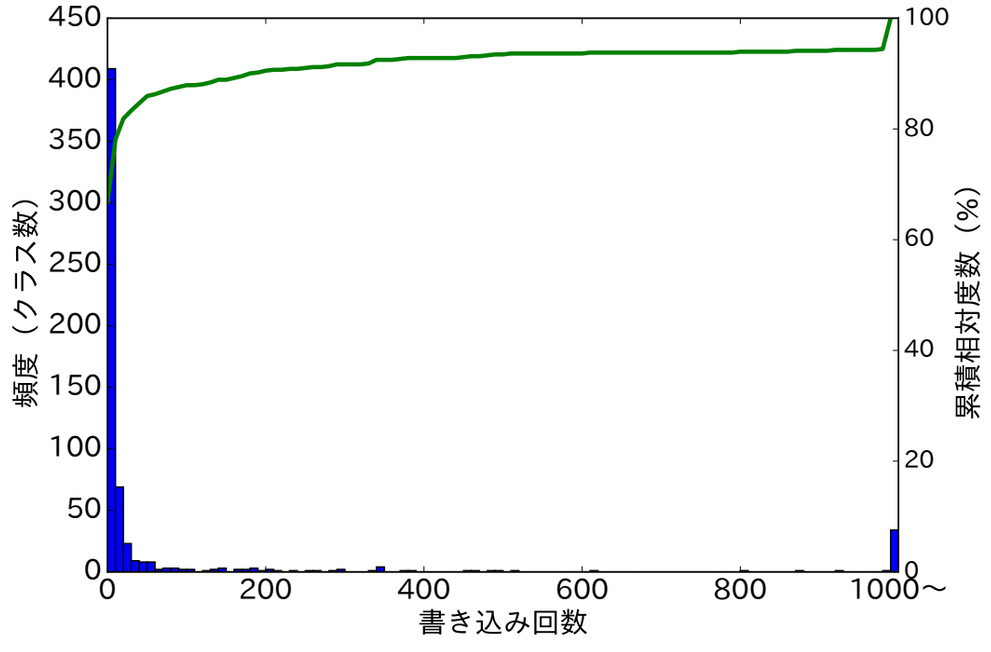

図 3.27　クラス単位での書き込みアクセス回数の分類（xalan）

3.5.3　書き込みアクセス回数による評価

図 3.28 は dacapo に含まれるベンチマークを実行した際の，メモリに配置されたオブジェクトへの書き込みアクセスの合計回数を示したものである．縦軸は対数表示である．分析の結果，NVM への書き込み回数は DRAM への書き込み回数と比較して最大で 11.8%，最小で 0.2% であった．この結果から，提案手法により NVM への書き込みアクセスが大きく抑制できていることがわかる．

図 3.29 は，図 3.28 に示したデータから，新世代領域に配置されたオブジェクトへの書き込み回数を除いたときのオブジェクトへの書き込みアクセスの合計回数を示したものである．縦軸は対数表示である．つまり，この図は旧世代領域に配置されたオブジェクトに対する書き込みアクセスの合計回数を示している．提案手法は旧世代領域に配置されたオブジェクトを対象に書き込みアクセスのふるまいに基づいたデータ配置を行っているため，この結果に着目することで昇格時 NVM への書き込みアクセスがどの程度抑制されているのかがわかる．図 3.29 の結果では図 3.28 に示した結果に比べて，NVM への書き込みアクセスを抑制する効果が小さくなっているように見える．特にベンチマーク項目 bloat に関しては，NVM に対する書き込みアクセスの回数が DRAM に対する書き込みアクセスの回数を超過している．

この理由は旧世代領域における，DRAM，NVM へのデータ配置の大きさに差があるためである．第 3.4.7 節で述べたように，提案手法では DRAM の使用量が NVM の使用量の一定以下になるようにごみ集めを実施する．そのため NVM に配置されるオブジェクト数は DRAM のそれに対して大きくなる．本実験ではオブジェクトの配置もオブジェクトに対する書き込みとして計測しているため，配置されるオブジェクト数が多い領域に対する書き込みアクセスの回数は，配置されるオブジェクトが多いほど，その領域ごとの書き込みアクセスの合計回数は増加する．また，配置されるオブジェクト数が多いほど，いかにそれらのオブジェクトに対する書き込み回数が少なくとも，その配置領域ごとの書き込みアクセスは，配置オブジェクト数が少ない領域に比べて増加すると考えられる．

図 3.30 は，旧世代領域の DRAM，NVM それぞれの領域への書き込みアクセス回数の計測値を計測時点の領域のサイズで平均したものである．つまり，それぞれの領域全体の平均的なメモリ書き込み回数を示したものである．この結果からは，旧世代領域における NVM の書き込み回数も，新世代領域も含めた全体で評価した時と同様に，DRAM に対する書き込みアクセスの回数に比べて少ないことがわかる．分析の結果，最も抑制効果が少ない場合でも，NVM に対する書き込みアクセスの回数を DRAM に対する書き込み回数の 15.7% に抑制することができた．

図 3.31 から図 3.40 には，それぞれのベンチマーク項目ごとに，書き込み回数の変化を時系列表示した．この結果からも，プログラムの実行を通して，NVM への書き込みアクセスが DRAM に比べて抑制されていることがわかる．

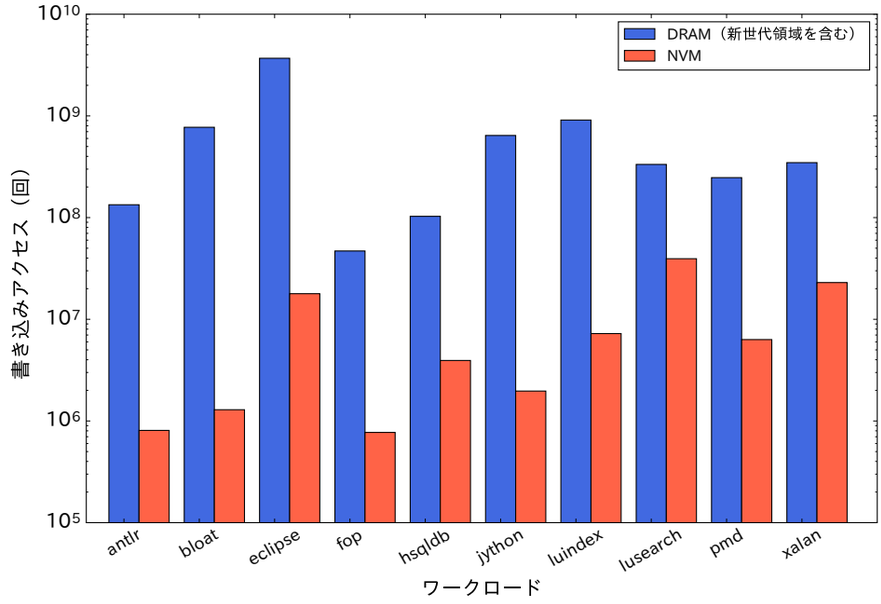

図 3.28　書き込みアクセス回数（新世代領域・旧世代領域）

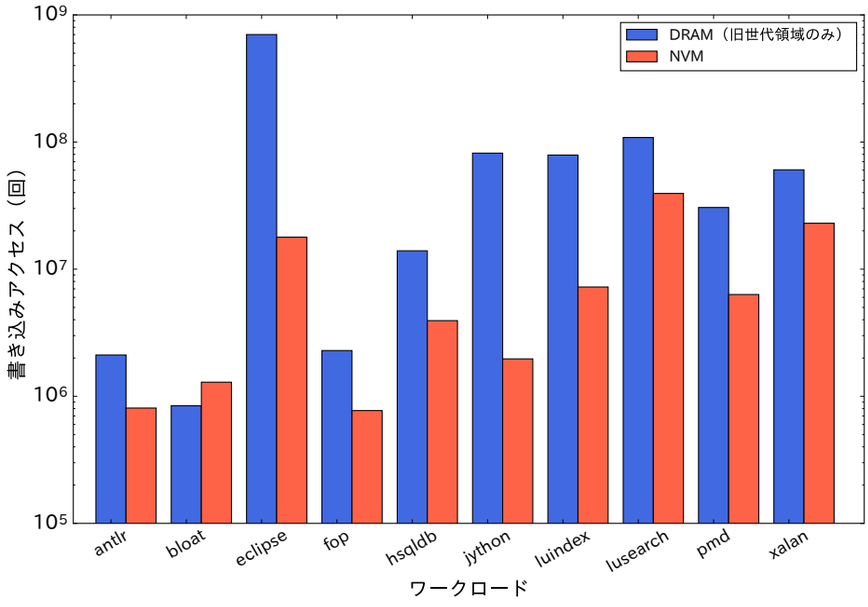

図 3.29　書き込みアクセス回数（旧世代領域のみ）

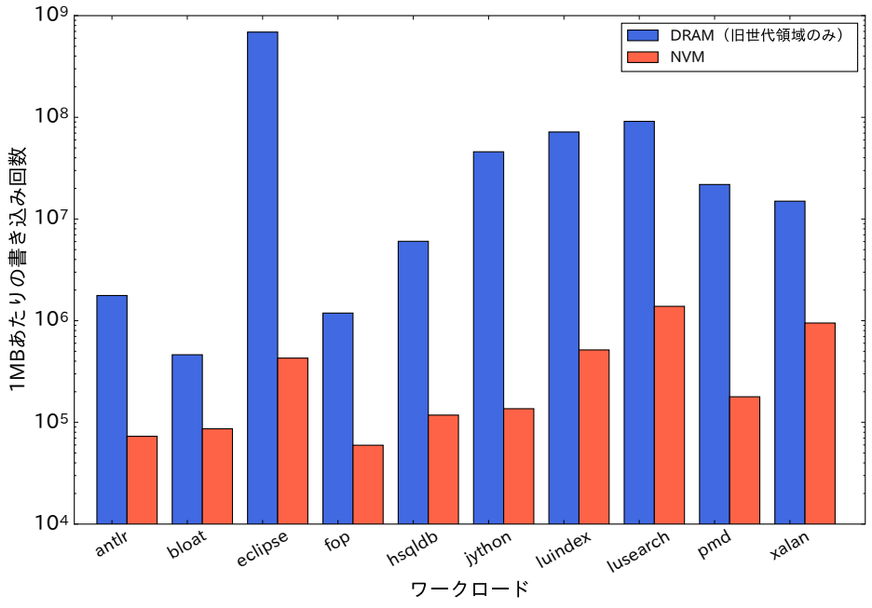

図 3.30　1MB ごとの書き込みアクセス回数（旧世代領域のみ）

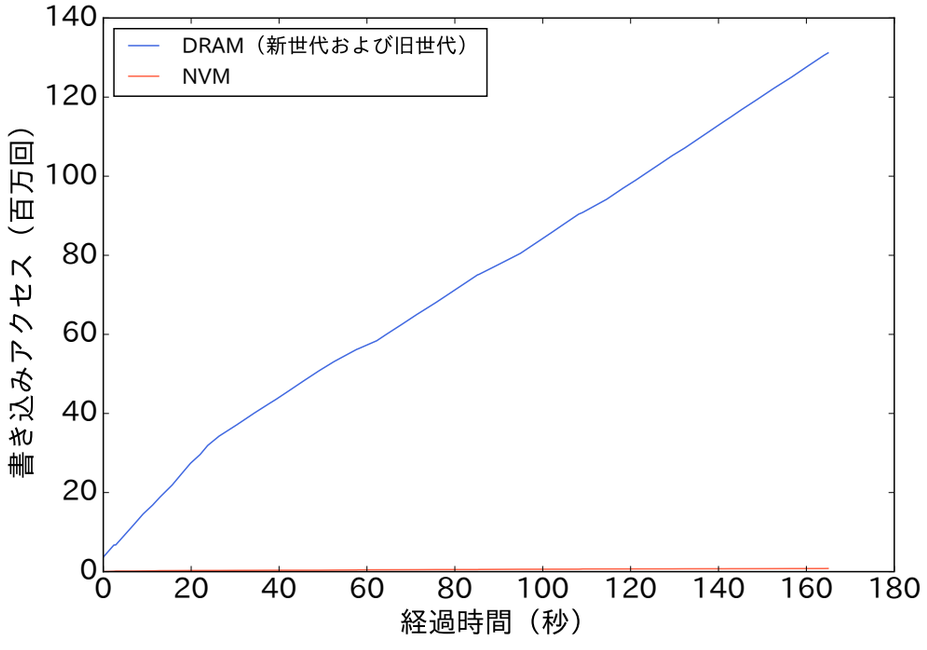

図 3.31　書き込み回数の変化（antlr）

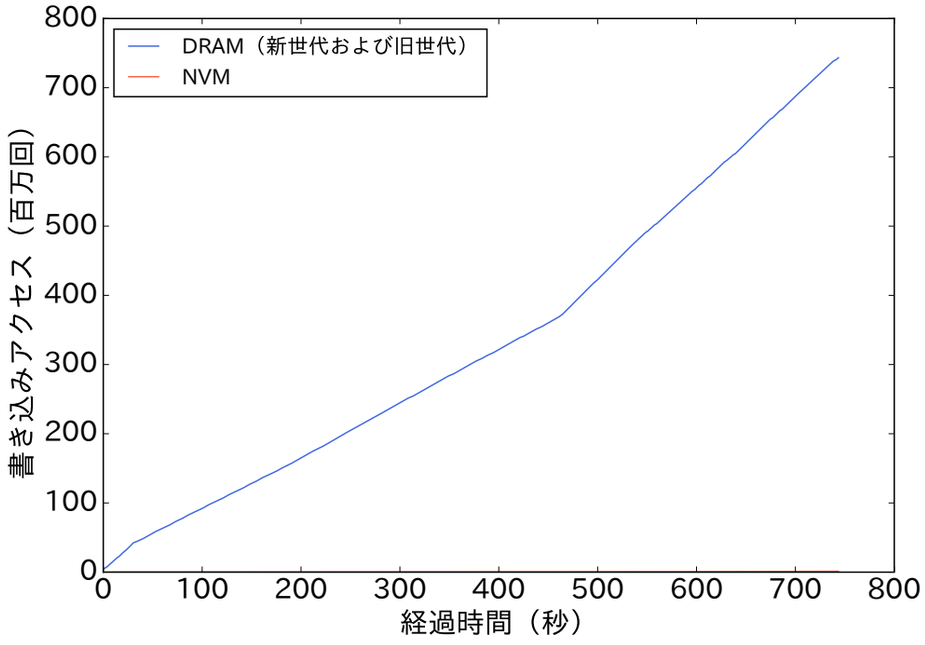

図 3.32　書き込み回数の変化（bloat）

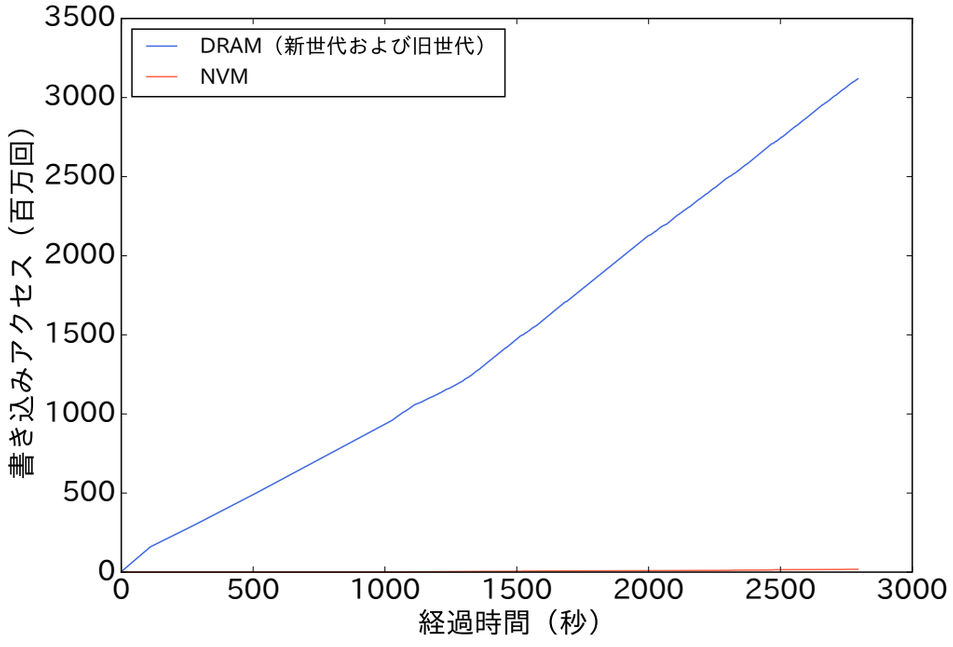

図 3.33　書き込み回数の変化（eclipse）

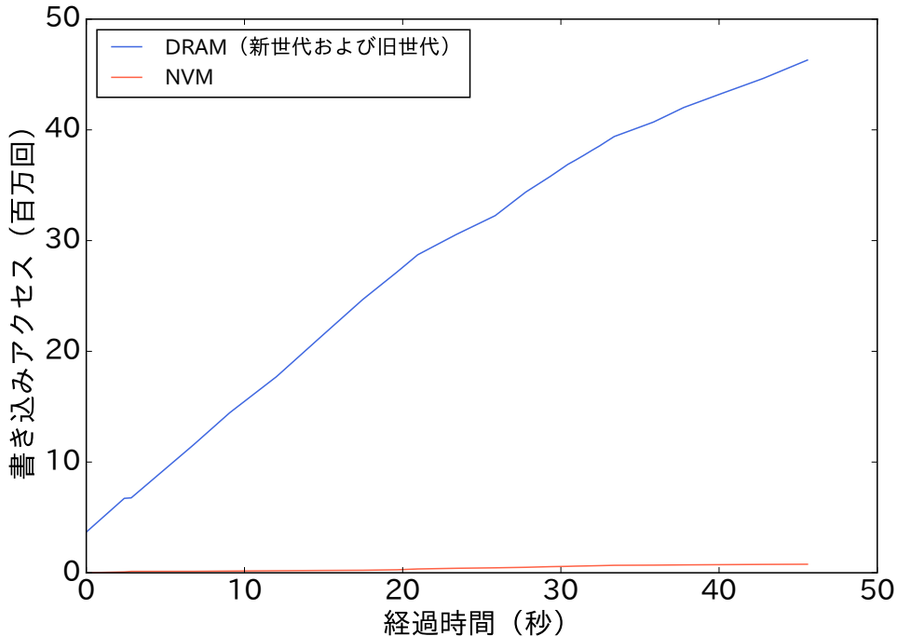

図 3.34　書き込み回数の変化（fop）

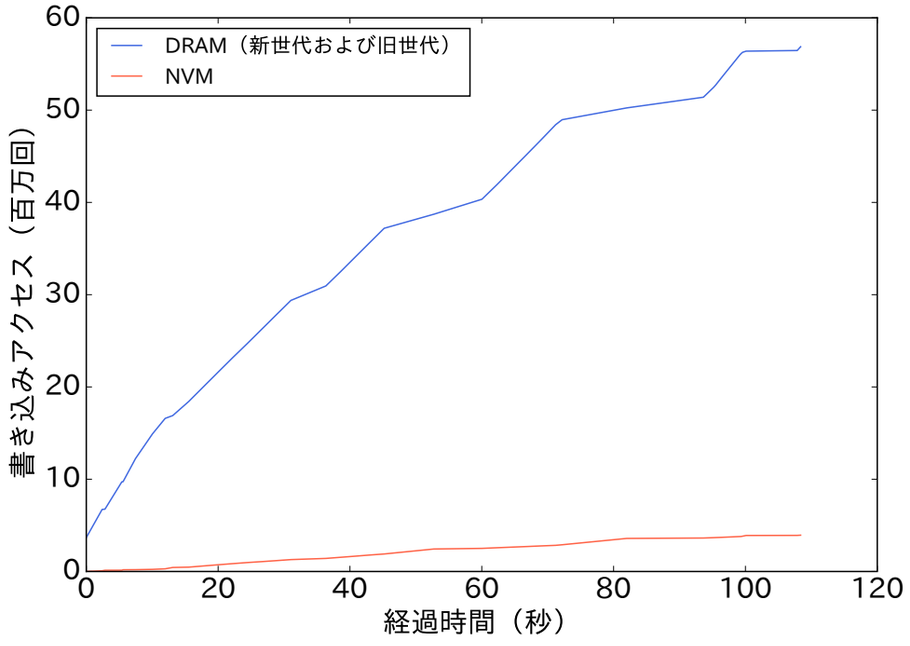

図 3.35　書き込み回数の変化（hsqldb）

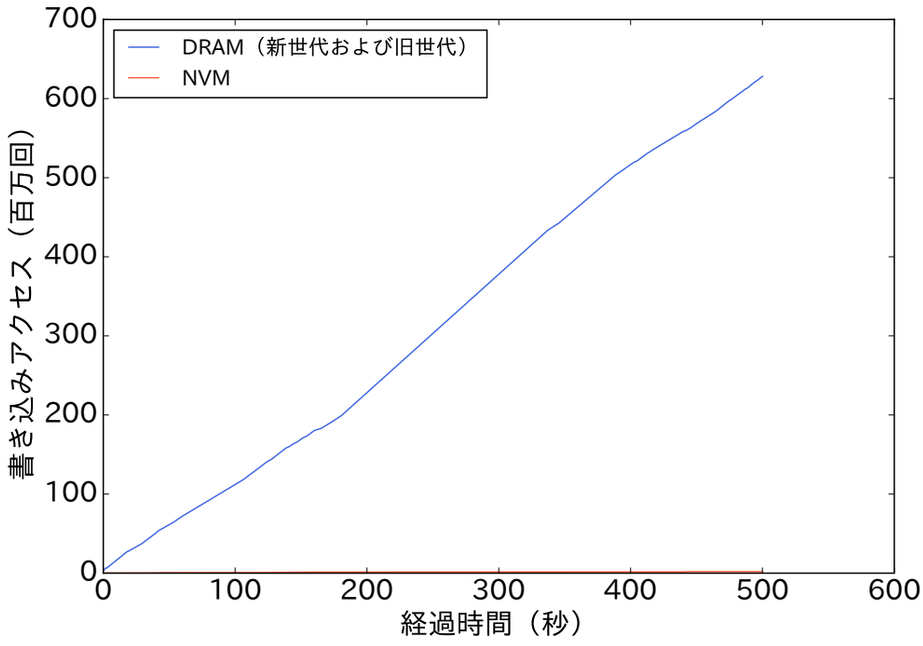

図 3.36　書き込み回数の変化（jython）

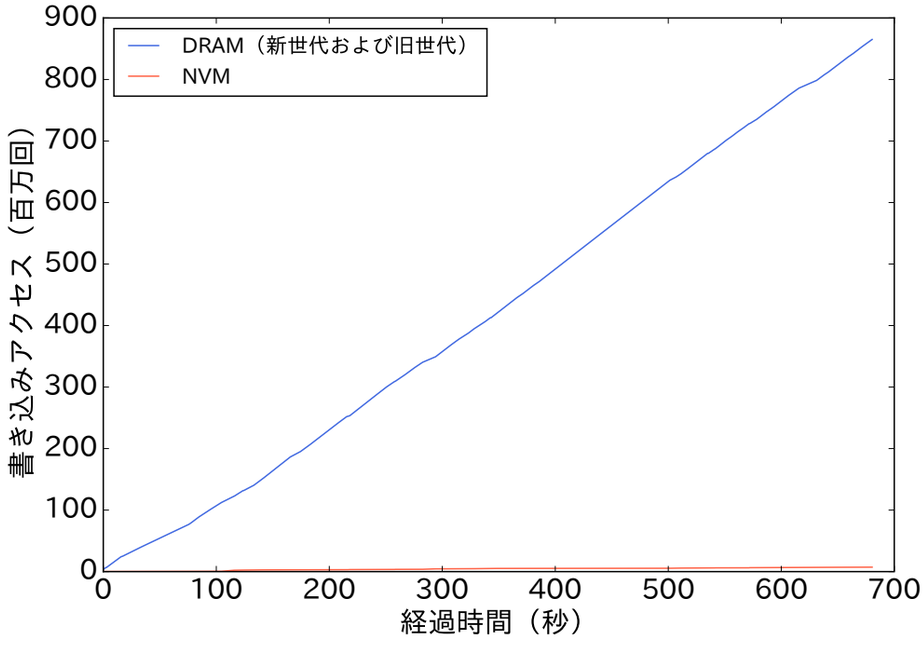

図 3.37　書き込み回数の変化（luindex）

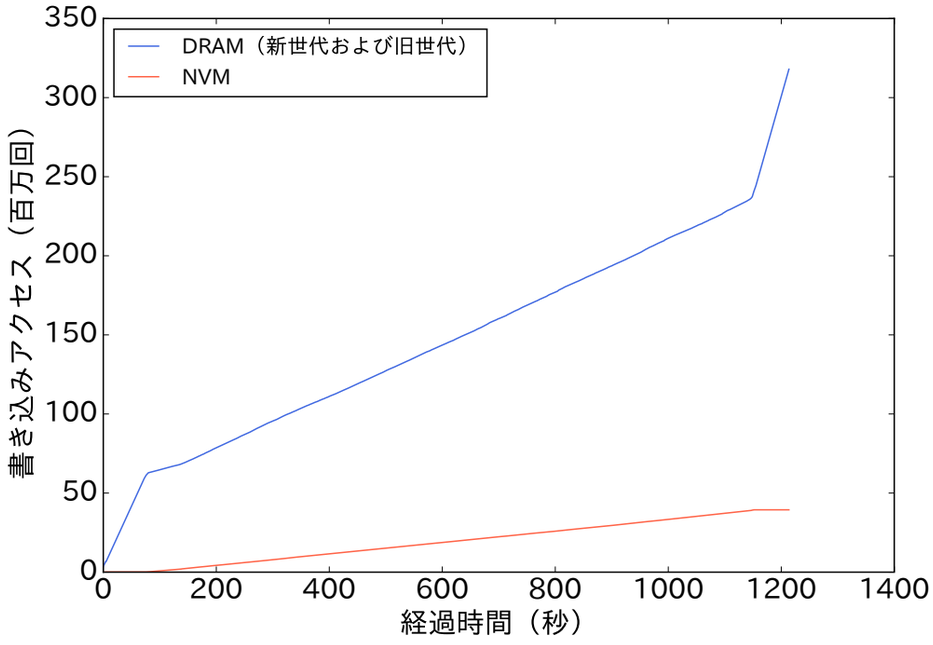

図 3.38　書き込み回数の変化（lusearch）

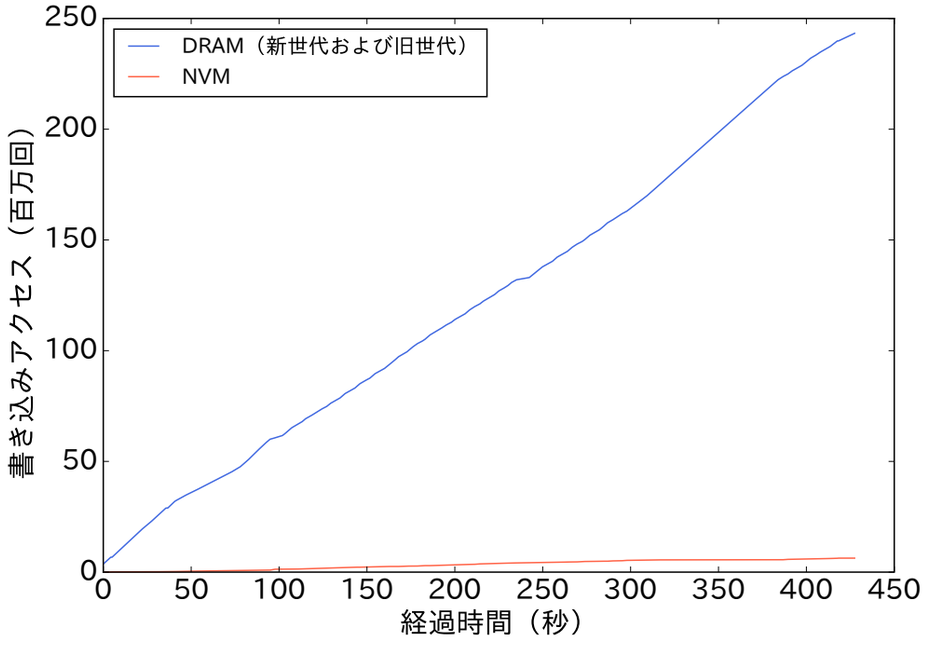

図 3.39　書き込み回数の変化（pmd）

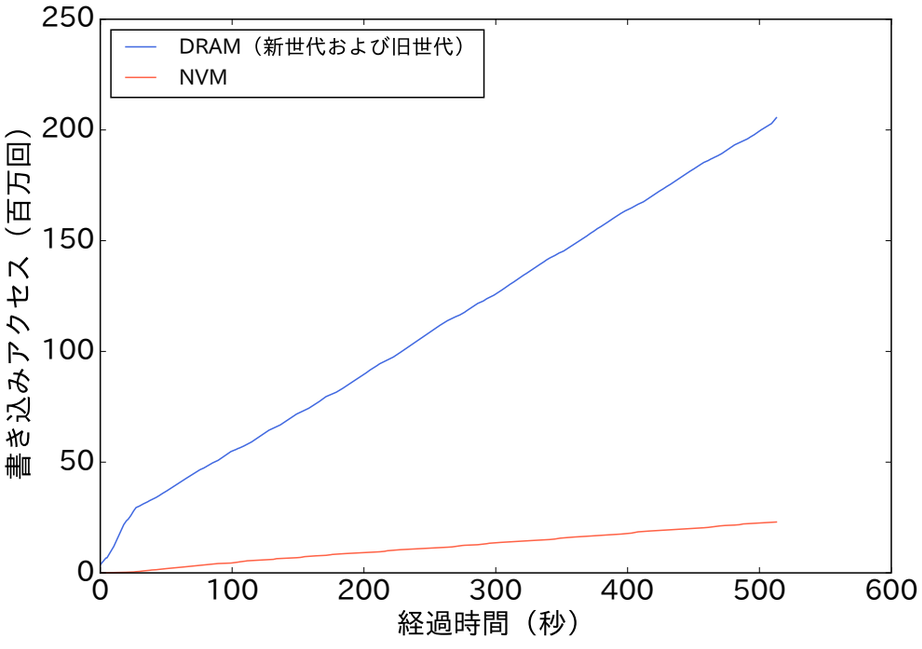

図 3.40　書き込み回数の変化（xalan）

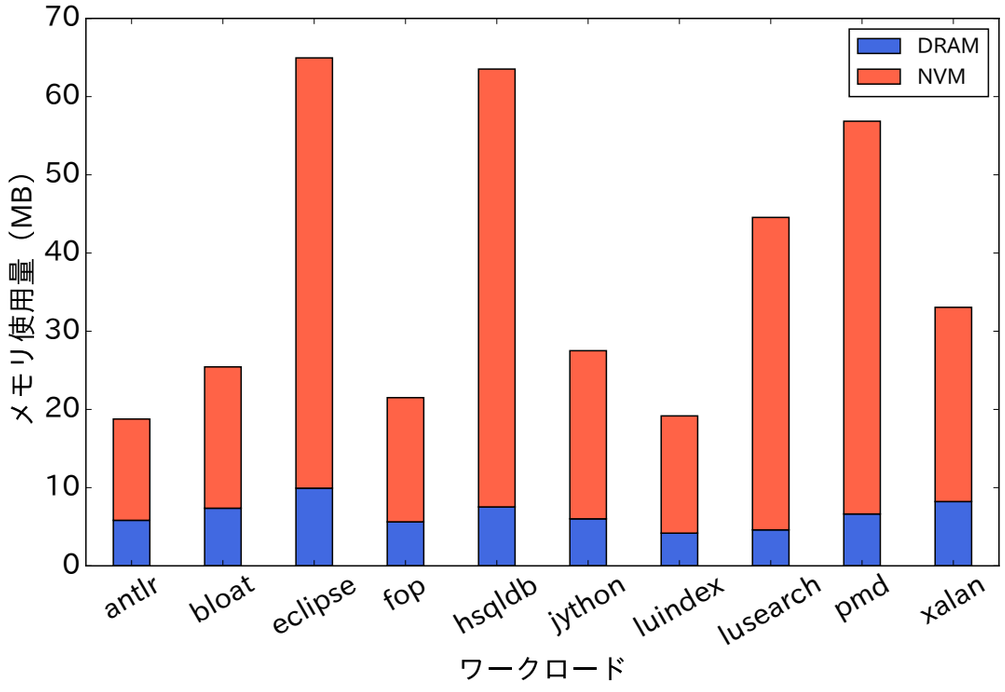

図 3.41　メモリ使用量

3.5.4　メモリ使用量による評価

図 3.41 はそれぞれのベンチマークを実行した際の，DRAM と NVM の使用量を示している．メモリ使用量は実行中に変化するが，ここでは一定周期で観測した値を算術平均したものを示している．この結果からは，いずれのベンチマーク項目についても，DRAM へのオブジェクト配置を抑制し，多くのデータを NVM に配置できていることがわかる．分析の結果，それぞれのベンチマーク項目のうち，もっとも DRAM の使用割合が多い場合でも全体の 31.0% であった．

図 3.42 から図 3.51 には，それぞれのベンチマーク項目ごとに，メモリ使用量の変化を時系列表示した．この結果からも，それぞれのベンチマーク項目の実行を通して，NVM に多くのオブジェクトを配置できていることがわかる．

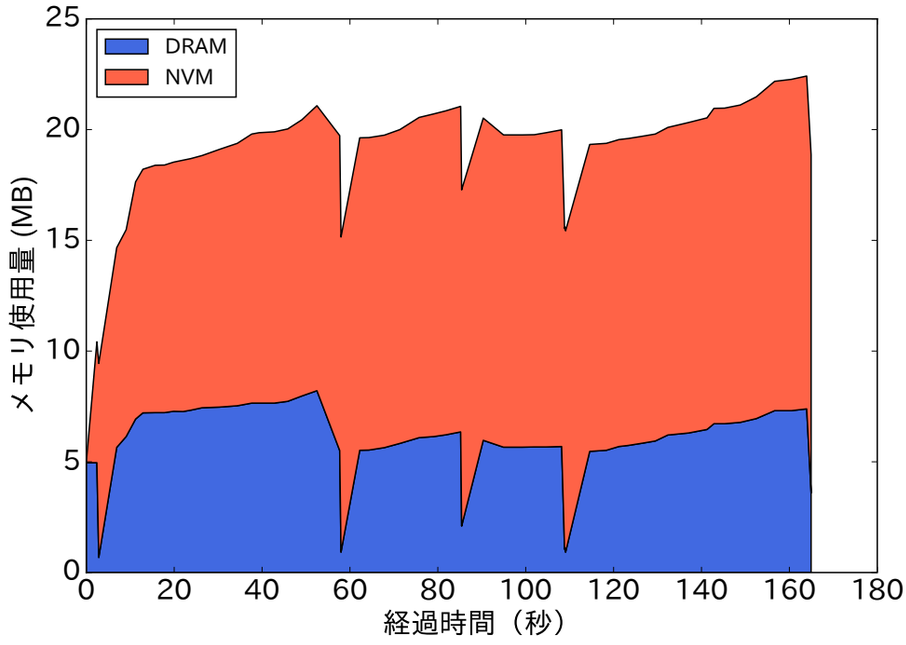

図 3.42　メモリ使用量の変化（antlr）

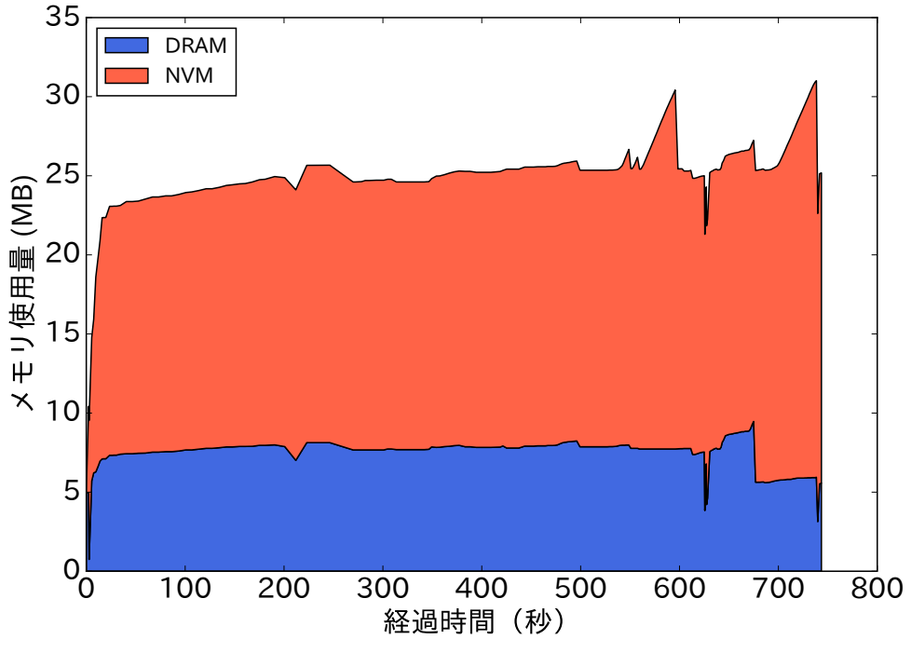

図 3.43　メモリ使用量の変化（bloat）

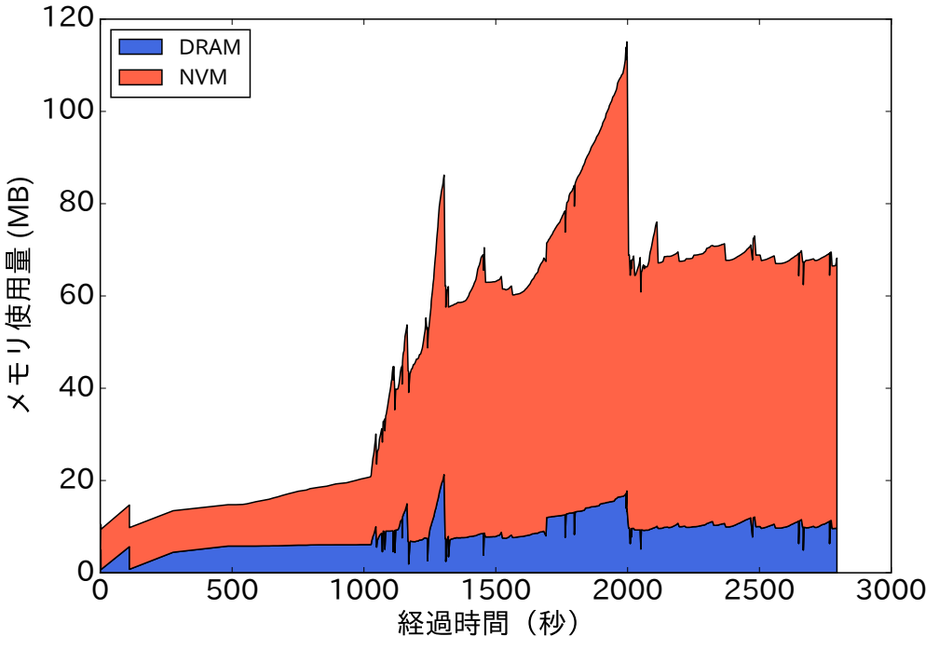

図 3.44　メモリ使用量の変化（eclipse）

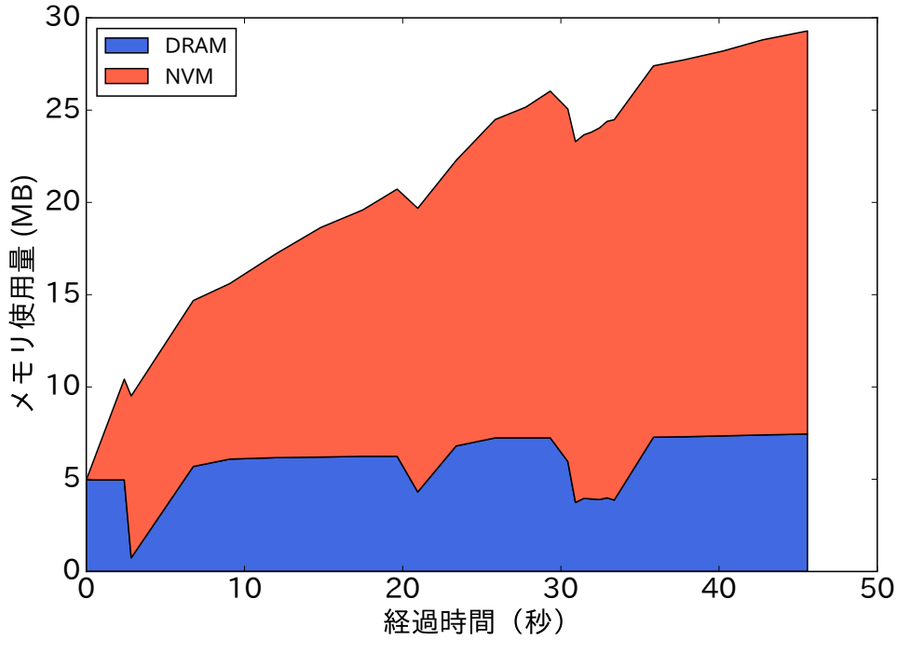

図 3.45　メモリ使用量の変化（fop）

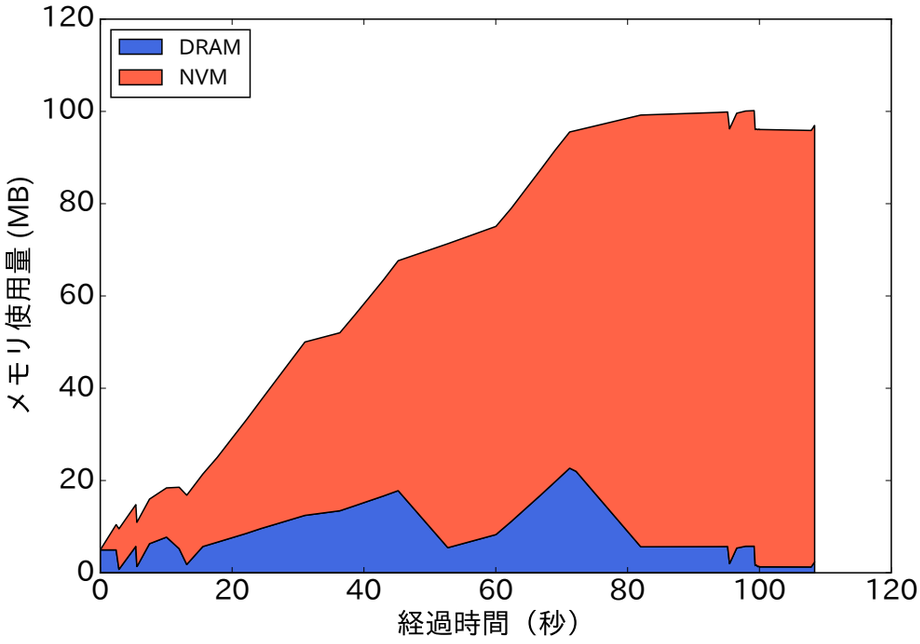

図 3.46　メモリ使用量の変化（hsqldb）

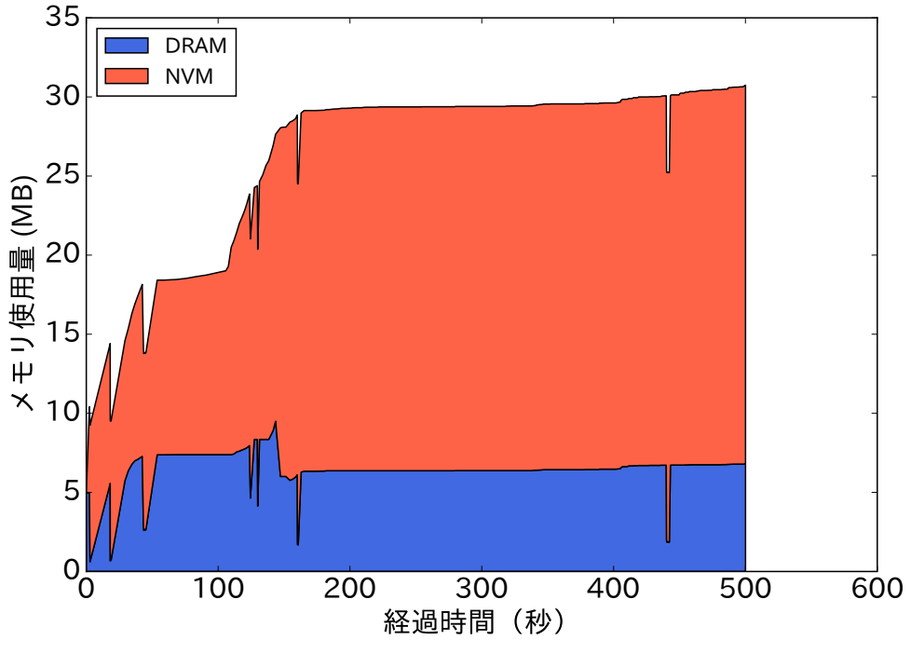

図 3.47　メモリ使用量の変化（jython）

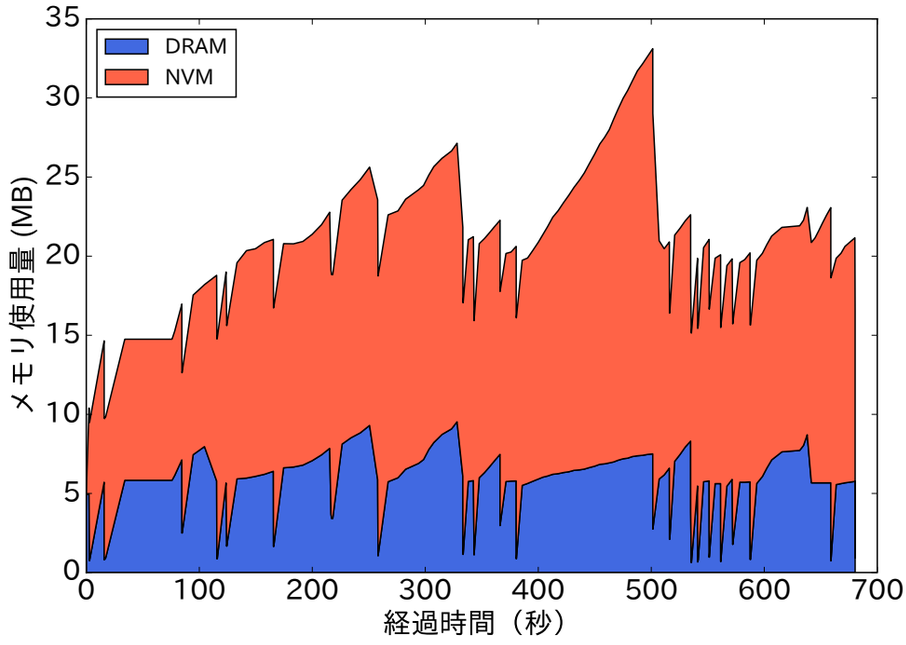

図 3.48　メモリ使用量の変化（luindex）

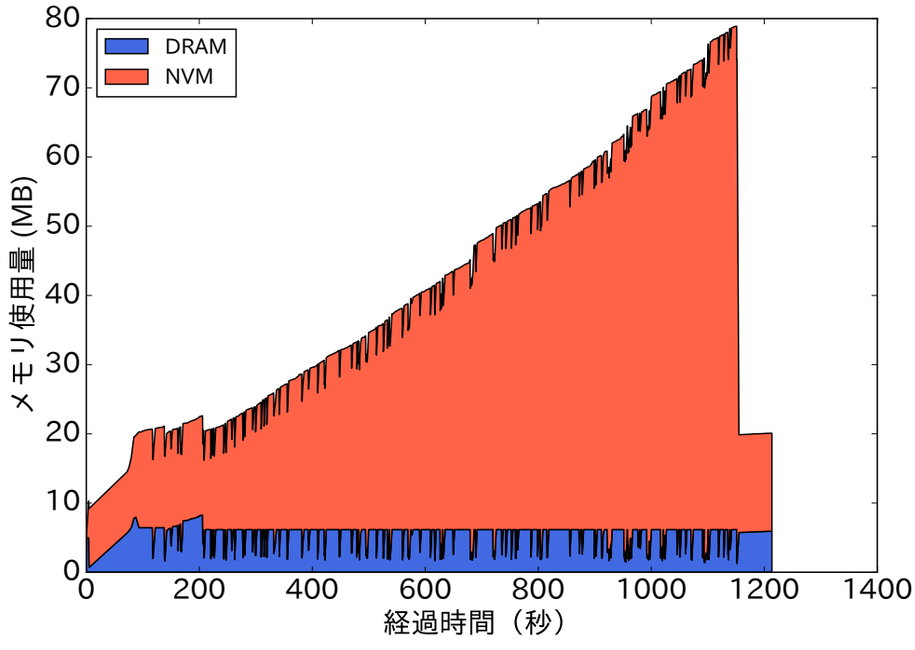

図 3.49　メモリ使用量の変化（lusearch）

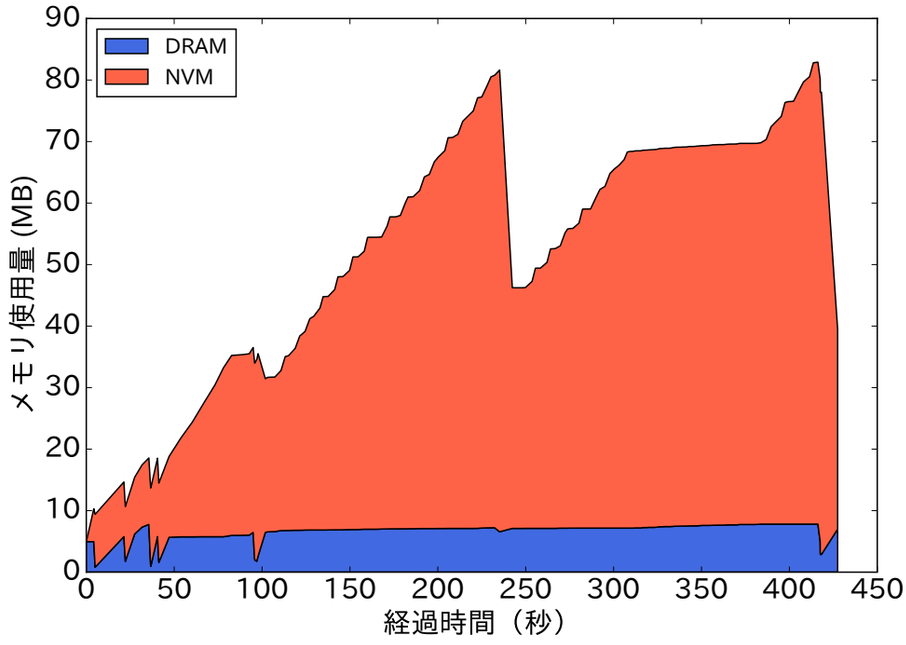

図 3.50　メモリ使用量の変化（pmd）

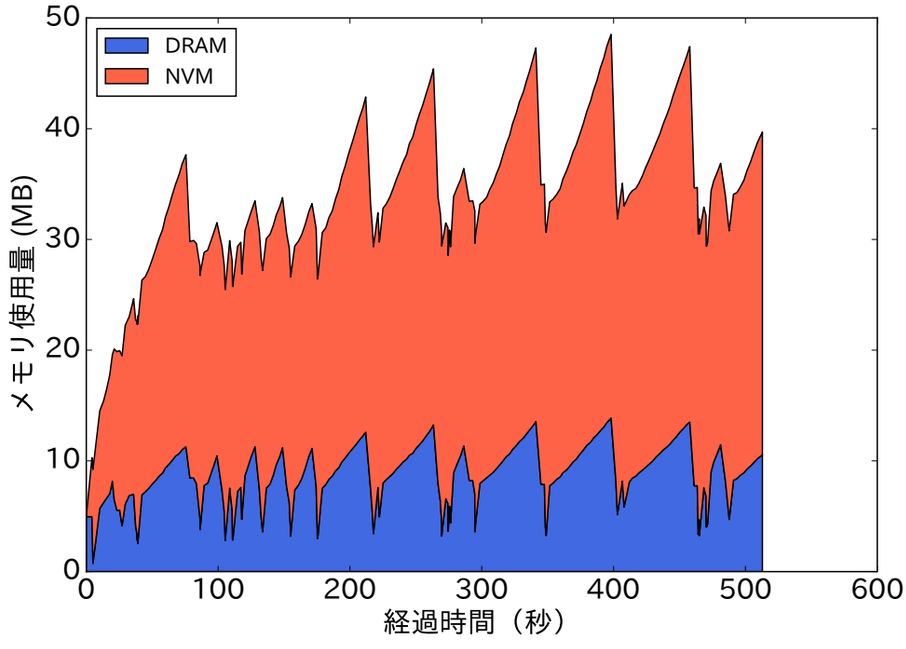

図 3.51　メモリ使用量の変化（xalan）

3.5.5　実行時間による評価

図 3.52 はそれぞれのワークロードを実行したときの実行時間を，提案手法を実装した前後で比較したものである．いずれのワークロードに関しても実行時間の増加が確認された．分析の結果，実行時間は最小で 2.1 倍，最大で 17 倍に増加しており，提案手法の実装が大きなオーバーヘッドを発生させていることがわかる．

この実行時間の増加の要因は大きく三つある．一つは，オブジェクトマイグレーションに伴うごみ集め処理の増加である．第 3.4.6 項で述べたように，提案手法では旧世代領域に配置されたオブジェクトの配置領域の見直しをごみ集め処理を利用して実施する．そのため，第 3.4.7 項で述べたように，通常のごみ集めのタイミングに加えて追加のごみ集めを旧世代領域で実施する．本実験で用いた実装では旧世代領域のごみ集め処理中はプログラムの動作が中断するため，ごみ集め処理の回数の増加が実行時間の増加につながっていると考えられる．

要因の二つめは，ライトバリア対象の追加である．第 3.4.4 節で述べたように，提案手法ではオブジェクトへの書き込み回数を計測するために，ライトバリア処理を利用する．本実験での提案手法の実装に利用した Jikes RVM は，性能向上を目的として，書き込み先のデータの種類に基づいて，ライトバリア処理の発生が最小限になるように工夫されている．提案手法の実装時には，全てのデータ書き込みを記録するために，この最小化されているライトバリア処理を全ての書き込みアクセス対象について有効にする．そのため提案手法の実装によりライトバリア処理の回数が増え，実行時間が増加したと考えられる．

要因の三つめは，本実験での提案手法の実装が実行性能の点で最適化されていないことである．本実験は，オブジェクト単位，セマンティクス単位での書き込みアクセスのふるまいの，ハイブリッドメモリアーキテクチャにおけるデータ配置に関する有効性を検証することを目的としている．そのため，実行性能よりもデータ配置の精度を高めることに重点を置いた実装となっている．例えば，前述したようにライトバリア処理の対象を拡大していたり，write-hot queue の処理が書き込みアクセスごとに呼び出すなど，データ配置の精度は高まるが実行時間が大きくなる実装となっている．

3.5.6　評価実験 1 のまとめ

第 3.5.3 項で示したように，提案手法を用いることで，NVM への書き込みアクセスを DRAM への書き込みアクセスと比較して大きく抑制できることがわかった．また第 3.5.4 項で示したように，DRAM の使用量を最大で 31% に抑制することができた．以上の結果より，提案手法を用いることで，DRAM へのデータ配置を抑制しつつ，NVM への書き込みアクセスを抑制できることがわかった．ハイブリッドメモリにおいては，DRAM の使用量は少なく抑えつつ，NVM への書き込みアクセスを抑制することが望ましいため，提案手法はハイブリッドメモリにおけるデータ配置に有用でことがわかった．

しかしながら一方で，第 3.5.5 項で示したように，提案手法の実装により，言語処理系の動作速度が低下することも明らかとなった．この原因は提案手法の実装によるごみ集め処理の増加や，精度の

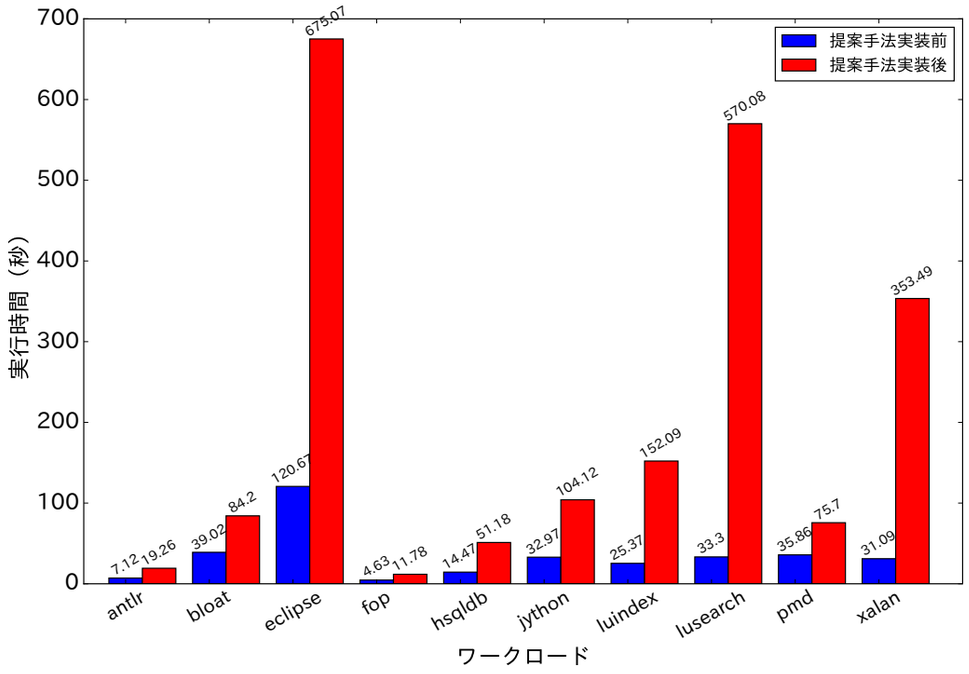

図 3.52　提案手法による実行時間の変化

高いデータ配置の決定を行うため処理の増加が原因であると推測される．提案手法を実用的に用いるには，データ配置の精度を落として動作速度を向上させることが必要である．データ配置の精度と実行性能はトレードオフの関係にあり，そのバランスを検討するのは今後の課題の一つである．

3.6　評価実験 2

第 3.5 節で述べた評価実験 1 では，提案手法の有効性を評価した．結果として，提案手法はハイブリッドメモリアーキテクチャにおける NVM への書き込みアクセスを大きく削減できることが明らかとなった．しかしながら評価実験 1 では，本章で着目している，データのセマンティクスごとの異なる書き込みアクセスのふるまいを用いた書き込みアクセスの予測が，その NVM に対する書き込みアクセスの削減に寄与しているのか，明らかになっていない．

本節では，評価実験 1 と同内容の実験を，セマンティクスを用いた書き込み予測が無効ある場合，有効である場合で実施し，その結果を比較する．その目的は，セマンティクスの書き込みアクセスのふるまいを利用した，書き込み多寡の予測が NVM への書き込みアクセスの削減に有用であるかを検証することである．‘

3.6.1 条件設定

本実験では，二つの実験条件を設定する（図 3.53）．条件 A はセマンティクスの書き込みアクセスふるまいに基づいたデータ配置の予測と振り分けが無効である条件である．この実験条件では，オブジェクトが新世代領域から旧世代領域に昇格する際に，NVM 領域への配置を行う．もし書き込みアクセスが集中するオブジェクトを検出した場合には，検出したオブジェクトを DRAM へ移動する．なお，条件 A の変種として，無条件に DRAM に配置し，書き込みアクセスの少ないオブジェクトを NVM に移動する条件も存在しうる．しかしながら，第 3.3 項や第 3.5.2 項で述べたように，今回対象とするプログラミング言語処理系では，生成されるオブジェクトの大部分が書き込みアクセスの少ないオブジェクトである．よって無条件に DRAM に配置すると，NVM へのオブジェクトマイグレーションが多発する．この状況はハイブリッドメモリの管理という面で不適切であるため，今回の実験では除外した．

条件 B はセマンティクスの書き込みアクセスのふるまいに基づいたデータ配置が有効である条件である．この条件での言語処理系の動作は評価実験 1 と同様である．オブジェクトの昇格時に，その所属するクラスに基づいて DRAM と NVM への間で配置の振り分けが行われる．NVM に配置されていなかがら書き込みアクセスの多いオブジェクトは DRAM に移動され，DRAM に配置されていながら書き込みアクセスの少ないオブジェクトは NVM に移動される．

3.6.2 実験結果

図 3.54 に，条件 A，B での NVM への書き込みアクセス回数を示した．縦軸は対数表示である．このグラフが示すように，すべてのワークロードについて，セマンティクスに基づく振り分けを行った場合に NVM への書き込み回数が削減できていることがわかる．分析の結果，最大で 63.9% の削減に成功した．

図 3.55 に，条件 A，B での実行時間を示した．このグラフが示すように，すべてのワークロードについて，セマンティクスに基づく振り分けを行った場合に，実行時間を削減できていることがわかる．分析の結果，条件 A での実行時間に対して，条件 B では最大で実行時間を 72.8% に抑制することに成功した．最も抑制効果が小さいワークロードでは，条件 B での実行時間は条件 A の 0.3% であった．

図 3.56 に，条件 A，B でのオブジェクト移動処理の回数を示した．第 3.4.6 項で述べたように，提案手法では旧世代領域のごみ集め処理のタイミングで旧世代領域の NVM，DRAM 間でオブジェクトの移動を行う．図はそのオブジェクト移動が伴うごみ集めの実行回数を示したものである．このグラフが示しているように，条件 B ではオブジェクトの移動を伴うごみ集めの回数が条件 A に比べて少ないことがわかる．これは条件 B では，セマンティクスの書き込みアクセスのふるまいに基づいた書き込み多寡の予測が適切に働いており，書き込みの多いオブジェクトを昇格時に DRAM に配置できているためだと考えられる．

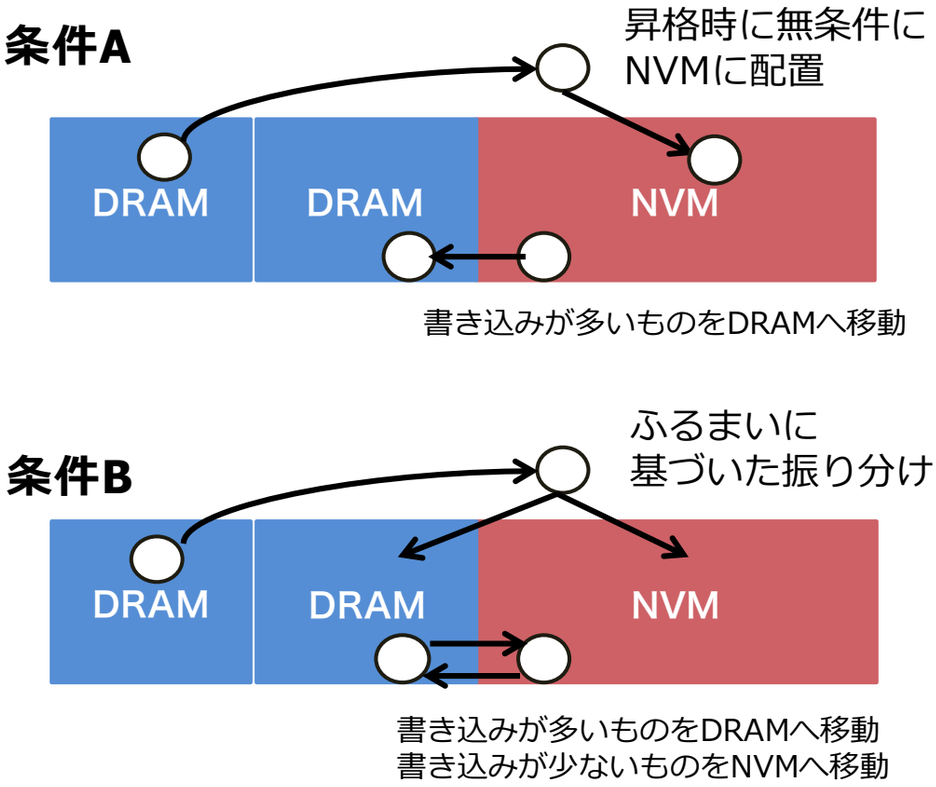

図 3.53　評価実験 2 での条件設定

3.6.3　評価実験 2 のまとめ

この評価実験では，評価実験 1（第 3.5 節）での実装（条件 B）に対して，それから昇格時における NVM，DRAM 間でのオブジェクト配置の振り分けを無効にした条件（条件 A）を設定して，その二条件での提案手法の効果を比較した．この目的は，セマンティクスごとに異なる書き込みアクセスのふるまいを用いた書き込みアクセス傾向の予測がハイブリッドメモリにおけるデータ配置に有効であるか否かを検証するためである．

図 3.54，図 3.55 に示した結果より，セマンティクスの書き込みアクセスのふるまいに基づいた予測を用いることで，NVM に対する書き込みアクセスと実行時間が削減できることがわかった．また図 3.56 に示した，オブジェクトマイグレーションを伴うごみ集め処理の回数の比較より，ふるまいに基づいた予測により，オブジェクトマイグレーションのためのごみ集め処理回数の削減も確認できた．

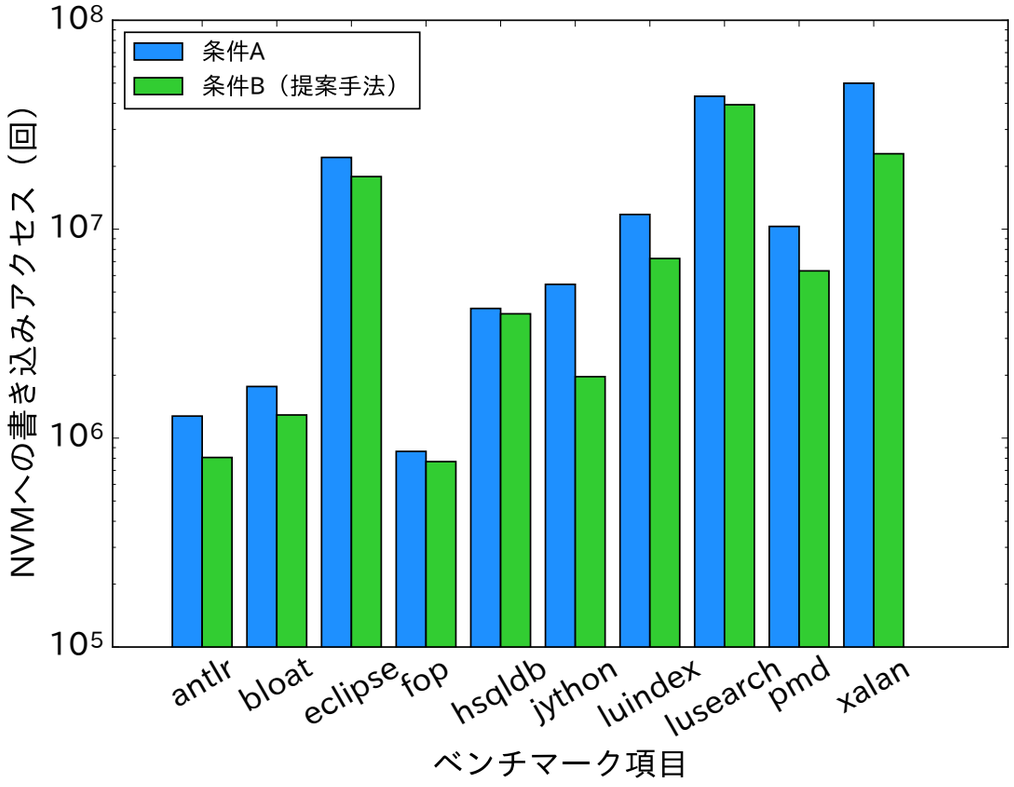

図 3.54　書き込みアクセス回数の比較（評価実験 2）

3.7　まとめ

本章では，第 1.1.1 項で述べた，DRAM と NVM によるハイブリッドメモリにおけるデータ配置の問題に関して，セマンティクスの書き込みアクセスのふるまいを利用して解決することを試みた．第 3.3 項，第 3.5.2 項で示したように，プログラムが生成するデータは，その書き込みアクセスの頻度に明らかな偏りがある．また，プログラミング言語処理系のレベルで得られるセマンティクスにも書き込みアクセスの頻度に偏りがある．この偏りを用いれば，データが生成されたタイミングで，そのデータに関する将来の書き込み傾向を予測することができる．この着想に基づき，オブジェクト指向言語処理系のレベルで，データの書き込み傾向を考慮したメモリ配置を決定する手法を設計，実装した．

また，二つの実験を通して提案手法の有効性を確認した．評価実験 1 では，提案手法がハイブリッドメモリにおける NVM への書き込みアクセスの抑制にどの程度効果があるのかを検証した．結果として，提案手法を用いることによって，多くのデータを NVM に配置しながら，NVM への書き込みアクセスを抑制できることがわかった．評価実験 2 では，評価実験 1 で確認できた提案手法の効果のうち，セマンティクスの書き込みアクセスのふるまいがどの程度寄与しているのかを検証した．実験

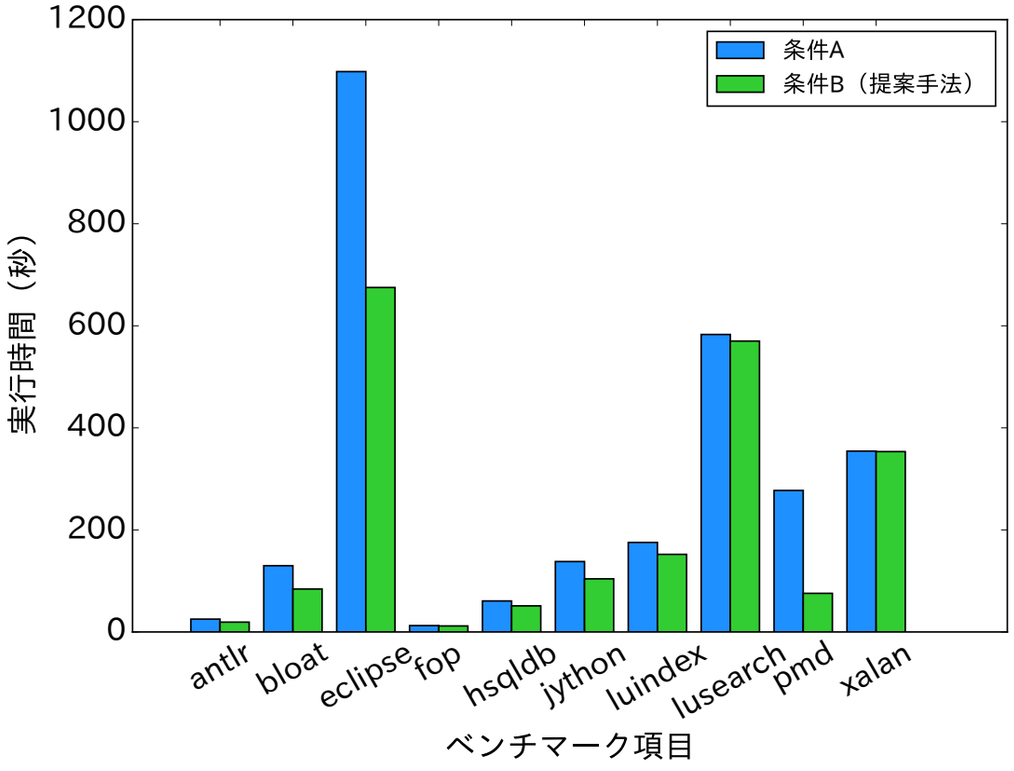

図 3.55　実行時間の比較（評価実験 2）

は，セマンティクスの書き込みアクセスのふるまいに基づいた書き込みアクセス頻度の予測を行った場合と，それを無効にした場合で，データの配置し，それぞれの領域への書き込みアクセスの回数や実行時間などを評価した．結果として，ふるまいに基づいた書き込みアクセスが，データの書き込みアクセス頻度の予測に有用であることが確認できた．

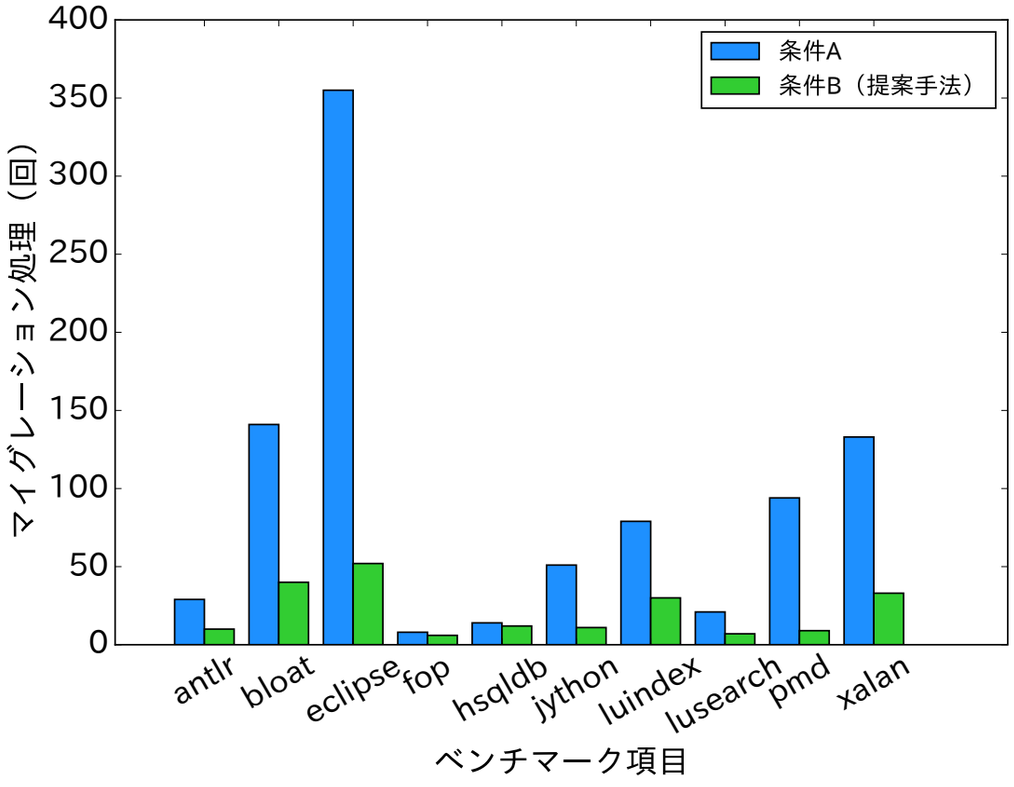

図 3.56　オブジェクトマイグレーション回数の比較（評価実験 2）

第 4 章

プロセスのリソース隔離

第 1.1.2 項で述べたように，Web アプリケーションプラットフォームにおけるメモリリソース管理には解決すべき問題がある．これまでの OS レベルでのメモリリソース管理では，直近の利用状況にのみ基づいてメモリの利用制限を行っていた．しかしながら，Web アプリケーションの複雑化，PaaS のようなサービスの管理者とプラットフォームの管理者が分離しているシステムモデル，仮想化インスタンスの集密化などを背景として，その従来のメモリ管理手法では，メモリリソースにまつわる問題に十分に対処できない．問題の解決には，実際に消費した量に基づいたメモリリソース管理だけではなく，メモリ消費の予測を考慮した管理手法が必要である．

本章では，この問題にプロセスのメモリリソース消費のふるまいを用いて解決を試みる．メモリリソース消費のふるまいは，プロセスやプロセスグループ単位でのメモリリソース消費の傾向である．このメモリリソース消費のふるまいには，直接的なものと，間接的なものがある．直接的なメモリリソース消費のふるまいは，プロセスごとの単位時間あたりのメモリ使用量の増分である．間接的なメモリリソース消費のふるまいは，それそのものはメモリ消費ではないが，間接的にメモリ消費が伴うようなイベントの発生傾向である．例えば単位時間あたりの子プロセスの増分は間接的なメモリリソース消費のふるまいである．なぜなら，子プロセスの生成にはメモリ消費の拡大が伴うからである．

ここでは，プロセスのメモリリソース消費のふるまいを利用したメモリ管理手法として，プロセスのリソース隔離を提案する．この提案手法では，異常なメモリ消費を引き起こす可能性があるプロセスやプロセス群を，そのメモリリソース消費のふるまいを用いて検出する．検出したプロセスには，通常のリソース制限に加えて，より厳しいリソース制限が課される．この状態をリソース隔離と定義する．リソース隔離されたプロセスは新たに追加されたリソース制限を受けながら実行を継続する．

4.1　メモリ消費のふるまいと異常検出

プログラムは，ときに意図しない急速で大量のメモリ消費を引き起こす．管理者はシステムで動作する全てのプログラムのメモリ使用の挙動を把握しているわけではない．マルウェアや悪意のある一般ユーザにより，大量のメモリ消費が引き起こされる可能性がある．また，システムで正当に利用されるソフトウェアに，メモリ管理に関する瑕疵があった場合も，大量のメモリ消費が発生することが

ある．悪意のないユーザも，プログラミングミスにより，同様の状況を引き起こす可能性がある．このような意図しない急速で大量のメモリ消費が発生する可能性は常に存在している．

このような問題のあるメモリ消費は，正当なプロセスの動作妨害や，システムの不安定化を引き起こす．メモリが不足することによって，新たなプロセス生成の失敗や動作中のプロセスの異常終了が起こる．このプロセス実行の妨害によって，正当なサービスの提供が阻害される．2 次記憶へのスワップを利用するシステムの場合は，大量のページイン処理，ページアウト処理によってシステムの応答性能やストレージの応答が低下することも考えられる．

このような問題のあるメモリ消費に対して，一般的には，使用できるメモリリソース量に上限を設けることで，これに対処する．プロセス単位やプロセスの種類ごとに使用できるメモリの量の上限を設定することで，たとえ異常なメモリ消費が発生したとしても，その使用量は設定値以下に抑制することができる．最も単純な戦略は，全てのプロセスに対して一律，または個別に上限を設定することである．また，任意のプロセス群ごとに上限を設定する戦略もある．たとえば，特定の Web アプリケーションを提供するための Web サーバ，アプリケーションサーバ，データベースサーバを 1 つのリソース制限の単位として登録することで，そのサービス提供に使用できるメモリ量の総量を制限することができる．

しかしながら，このメモリの使用量のみに基づいたリソース制限には二つの問題がある．問題の一つは，実際にメモリの浪費が起こってからしか対策が実施できないことだ．このメモリ使用量の制限は，制限対象のメモリ使用量が制限値を超えた時に初めてメモリの割り当てを制限する．そのため，実際に制限値を上回るメモリの要求が発生するまでは問題に対処できない．設定した制限値までは，メモリの浪費が発生し，システムの空きメモリ領域が減少する．オペレーティングシステムの実装によっては，空きメモリ領域をブロックデバイスのキャッシュとして用いるため，この空きメモリ領域の現象により，キャッシュの容量減少や，メモリ確保のために，キャッシュのライトバックが発生する．これはファイルアクセス性能の低下につながる．

二つ目の問題は，メモリの利用効率が悪くなることだ．メモリ使用量の制限のみで問題に対処するためには，そのシステムでのメモリ使用量の上限値の合計が，システムで利用可能なメモリの総量より少ないことが求められる．もしシステムでのメモリ使用量の上限が，システムで利用可能なメモリの総量より多い場合は，問題となる事象を完全に防ぐことができない．たとえば，512MB のメモリ空間があるシステムで，プロセス A, B, C が動作すると仮定する．なお議論の単純化のために，ここで言及システムにはスワップ空間がなく，またカーネルの動作に必要なメモリは 0 であると仮定する．プロセス A，B，C にはそれぞれ制限値 a，b， c を設定するものとする．この a，b，c の合計が 512MB 以下であれば，プロセス A，B，C がたとえすべて同時に異常なメモリ消費をしたとしても，メモリが不足することはない．しかしながら，a，b，c の合計が 512MB を超過すると問題に対処することができない．たとえば a，b，c それぞれに 300MB を制限として設定するとする．このときの制限値の合計は 900MB となり，搭載 RAM の量を超過する．この制限下で，プロセス A，B がそれぞれ 200MB メモリを使用しているとする．このときメモリ使用量の合計は 400MB である．この状況でプロセス C が異常なメモリ消費を引き起こすと，その制限値である 300MB よりも少ない，112MB を消費してところでシステムの空きメモリは枯渇する．しかしながら，a，b，c の合計を 512MB 以

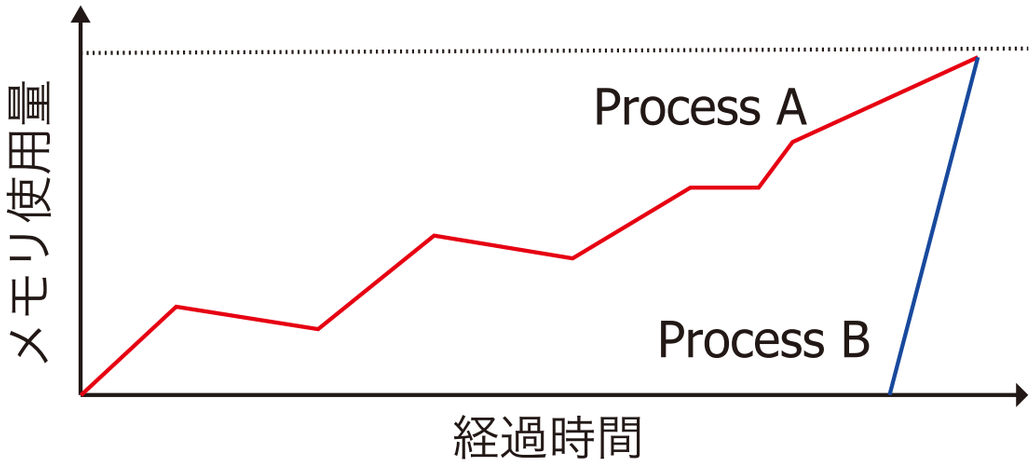

図 4.1　メモリ消費速度の違い

下に設定すると，メモリの利用効率が悪化する．プロセス A，B，C は，常にその制限値の上限までメモリを使用するわけではない．そのため，それぞれの制限値の範囲で，使用していないメモリ空間は他のプロセスが活用することができない．以上のように単純なメモリ使用量による制限だけでは，メモリ利用の効率を高めながら，問題のあるメモリ消費行動を防止することができない．これに対処するためには，新たなメモリ管理の枠組みが必要である．

この問題を解決するためには，メモリ消費の使用量に加えてメモリ消費のふるまいを利用することが有効である．メモリ消費のふるまいとは，メモリ消費やメモリ消費を引き起こすイベントの経過情報である．例の一つとして，メモリ消費の増加速度をここでは議論する．メモリ消費の増加速度は，プロセスごとの，単位時間あたりのメモリ消費の増加量である．図 4.1 に示すように，同じメモリ消費量でもそこに至るまでの消費速度はプロセスによってことなる．緩やかなものもあれば，急激なものもある．この経過情報の差を利用することで異常なメモリ消費の兆候を検出し，大量のメモリ消費が実際に発生する前に問題に対処することが可能である．

メモリ消費のふるまいは，メモリ消費の増加速度に限らない．メモリ消費を間接的に引き越すイベント発生の経過情報もメモリ消費のふるまいとして利用可能である．例えば，プロセス生成の頻度は，メモリ消費のふるまいとして利用できる．プロセスの生成が起こると，新しいプロセスについてメモリ空間が割り当てられ，間接的にシステムのメモリ使用量の増加を引き起こす．この性質を悪用して，意図的にプロセスを大量生成し，メモリ枯渇を引き起こす攻撃手法がある（fork 爆弾攻撃）．この攻撃に伴うメモリ消費の急増も，プロセス生成の頻度を監視することで，未然に検出することができる．

4.2　メモリ消費のふるまいに基づいたリソース隔離

第 4.1 節で述べたように，意図しない急激で大量のメモリ消費が引き起こす問題に，既存のメモリ使用量のみに基づいたリソース制限のみで対処するのは難しい．そこでこの問題に対して，本章ではメモリ消費のふるまいに基づいたリソース隔離を提案する．メモリの使用量を監視し，その違反時に

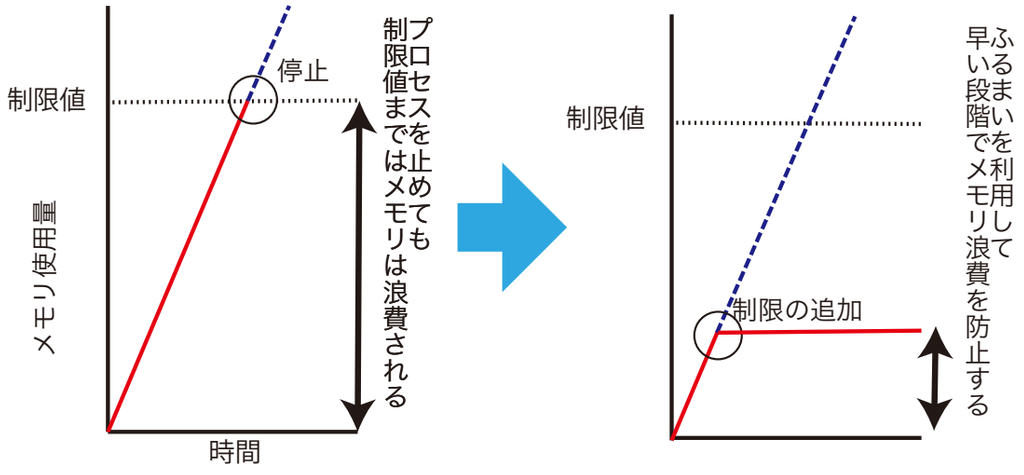

図 4.2　既存手法とリソース隔離（提案手法）の比較

対処する既存手法に対して，提案手法では，メモリ消費のふるまいに基づいて大量のメモリ消費の兆候を検出し，リソース制限を加え，その大量消費を未然に防止する（図 4.2）．

提案手法は二つの機能から構成されている．一つはメモリ消費のふるまいに基づいた，異常なメモリ消費の兆候の検出である．図 4.1 に示したように，プロセスのメモリ消費はその使用量だけでなく，メモリ消費の増加速度でも評価することができる．提案手法では，プロセスごとに単位時間あたりのメモリ消費増加量（メモリ消費速度）を計測し，それがしきい値を超過したプロセスを検出する．このしきい値については，対象システムの平常状態におけるメモリ消費速度を計測し，その最大値にマージンを持たせたものを設定する．

異常を検出した際には，問題を引き起こしているプロセスに対する新規のメモリ割り当てを拒否したり，そのプロセスを強制終了させる手段が最も単純な方法である．しかしながら，この方針には，偽陽性検出によって正常なプロセスが終了される問題がある．ふるまいに基づいたメモリ消費異常の検出には，偽陽性検出を起こす可能性がある．異常検出に用いるしきい値は，提案手法の対象システムごとに適切に設定されるべきであるが，計測時の想定漏れや，ソフトウェアの変化によって，正常なプロセスにもかかわらず，しきい値を超過するケースが想定される．そのため，異常なメモリ消費の兆候を検出したプロセスをその時点で停止すると，偽陽性検出した正常なプロセスの実行を中断する可能性がある．

提案手法では，検出時のプロセスの停止ではなく，厳しいメモリ利用制限を課すリソース隔離を実施する．メモリ消費のふるまいに基づいて，異常なメモリ消費の兆候ありと検出されたプロセスは，リソース隔離状態という特別な状態に設定される．この隔離状態にあるプロセス群は，使用できる物理メモリ量の合計が一定以下に制限される．その制限を超過した場合，もしシステムがスワップ領域があれば，リソース制限されたプロセス群はスワップ領域からメモリ領域を割り当てられる．もしスワップ領域を持たないシステムであった場合は，リソース制限されたプロセス群からのメモリの割り当て要求は失敗する．前述のとおり，メモリ消費のふるまいに基づいた，異常なメモリ消費の兆候の

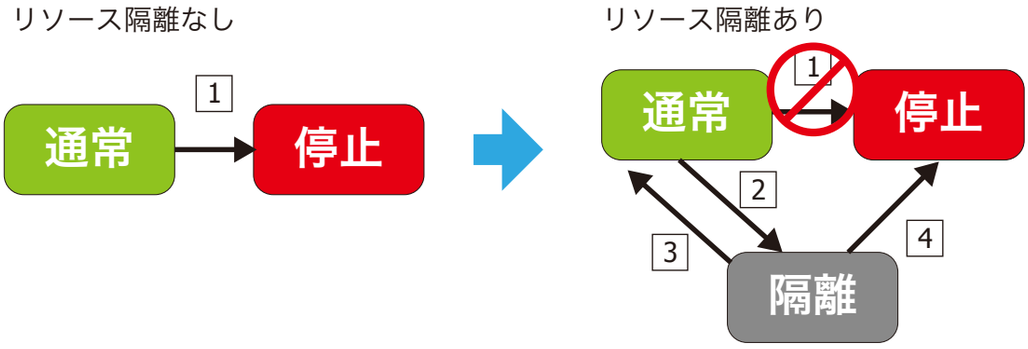

図 4.3　リソース隔離の状態遷移

検出は，偽陽性判定である可能性がある．そのため，リソース隔離の対象となったプロセスは定期的にそのふるまいを再検査され，もし，実際には大量のメモリを消費していないと判定された場合には，リソース隔離の対象から外され，メモリ使用量の制限が解除される．

図 4.3 に問題検出時にプロセスの実行を中断する方式（リソース隔離なし）と，提案手法のリソース隔離におけるプロセスの状態遷移を示した．リソース隔離なしの場合は，異常なメモリ消費の兆候を検出した時点でプロセスを停止する．リソース隔離がある場合には，兆候を検出した時点ではプロセスを停止せずに，隔離状態に設定し，メモリ使用量の制限を行う（図 4.3 2 ）．隔離状態に置かれたプロセス群は，その利用できるメモリ量の合計が一定以下に制限されるため，システム全体や，他の正常なプロセスに与える悪影響を抑制することができる．リソース隔離されたプロセスは一定周期ごとにそのふるまいを再検査され，実際には大量のメモリ消費を引き起こしていないと検出された場合は，隔離状態を解除され，通常状態に戻る（図 4.3 3 ）．これにより，仮にふるまいに基づいた異常なメモリ消費の検出が偽陽性であった場合も，プロセスを停止せずに実行を継続することができる．もし隔離状態のプロセス群が使用できるメモリ領域がスワップ領域も含めて枯渇した場合は，隔離状態のプロセス群は停止される．

4.3　提案手法の実装

提案手法の具体的な実装例として，メモリ消費のふるまいに基づいたリソース制限を行う機構を設計し，既存の OS に実装した．設計と実装の対象は Linux kernel 3.14.0 である．この節ではその設計と実装について述べる．

本機構は Web アプリケーションに対するリクエストを処理するプロセスのメモリ消費のふるまいを監視し，防御対象である Web アプリケーションのふるまいから逸脱するものを検出する．検出したプロセスは，異常なメモリ消費を引き起こすリクエストを処理しているものと判断され，そのプロセスが利用可能なメモリリソースの量が制限される．また，マルチコア 環境においては，その制限を受けたプロセスが実行される CPU が限定される． このリソース制限を受けた状態のプロセスを制限対象プロセスと呼ぶ．

リソース利用が制限されたプロセスは，使用していたメモリを解放する際にそのふるまいを検査され，異常なふるまいがなければリソース制限が解除される．以下では，本機構を構成する処理と対象 OS での実装について述べる．

4.3.1 メモリ消費のふるまいの算出

提案手法では，メモリ消費のふるまいとして，プロセスごとの物理メモリの消費速度に着目する．メモリの消費速度は，1 秒あたりの物理メモリの使用増加量とする．メモリ消費が減少傾向にあるときは，負の値を取る．図 5.2，5.3 に示したように，これらのふるまいは，正常なリクエスト処理時と DoS 攻撃を引き起こすリクエストの処理時とで，明らかに異なっている．Web アプリケーションへのリクエストを処理するプロセスについて，このメモリ消費のふるまいを監視することで異常なメモリ消費のふるまいを検出する．

このふるまいの計測は，プロセスごとのメモリページの割り当て処理のタイミングで行う．メモリページの割り当ては頻繁に起こるため，すべてのメモリページの割り当てタイミングではなく，一定の回数ごとに行う．この回数を MINFAULT_SAMPLE_RATE と定義する．計測は，計測点間での物理メモリ使用サイズの差分と計測時間の差分を用いて行う．この計測のために，OS のプロセス管理情報に物理メモリ使用量と，それを記録した内部時刻を記録する．このプロセス管理情報に記録された過去の物理メモリ使用量と OS カーネルの内部時刻と現在の状況との差分を取ることで，メモリ消費のふるまいを算出する．

この実現のため，実装対象 OS のページフォールト処理にメモリ消費のふるまいを計測する処理を追加した．またプロセス管理情報を記録する task_struct 構造体に，直近 1 回分の計測時刻と物理メモリ使用量のための領域を追加した．ふるまいの算出に用いる物理メモリ使用サイズは，プロセス管理情報から取得できる Resident Set Size (RSS) を用いた．また OS カーネルの内部時刻として Linux kernel の関数である ktime_get_ns() 経由で取得可能なナノ秒単位のモノトニック時刻を用いた．

4.3.2 メモリ消費のふるまいに基づいたリソース制限

提案手法は，メモリ消費のふるまいの算出時に急激なメモリ消費を行うプロセスを検出した場合，そのプロセスは問題のあるメモリ消費していると判断し，そのプロセスにリソース制限を課す．この検出は，算出したふるまいと基準値を比較して行う．あるプロセスについて，算出したメモリ消費のふるまいが基準値よりも大きければ，そのプロセスはリソース利用が制限される．この基準値を LIMIT_THRESHOLD と定義する．

制限を受けた複数のプロセス群は，それらのプロセスが利用できる物理メモリの総量を一定以下に制限される．この制限値を MEMORY_LIMIT とする．後述するリソース制限の解除に利用するため，制限対象となったプロセスの情報はプロセス管理情報とは独立したデータ構造に記録される．このデータ構造には，対象となったプロセスの情報や，制限を実施したときのメモリ消費のふるまいや，制限実施時の物理メモリの消費量を記録する．

これを実現するために，実装対象 OS のリソース管理機構である cgroups [56] を利用した．cgroups

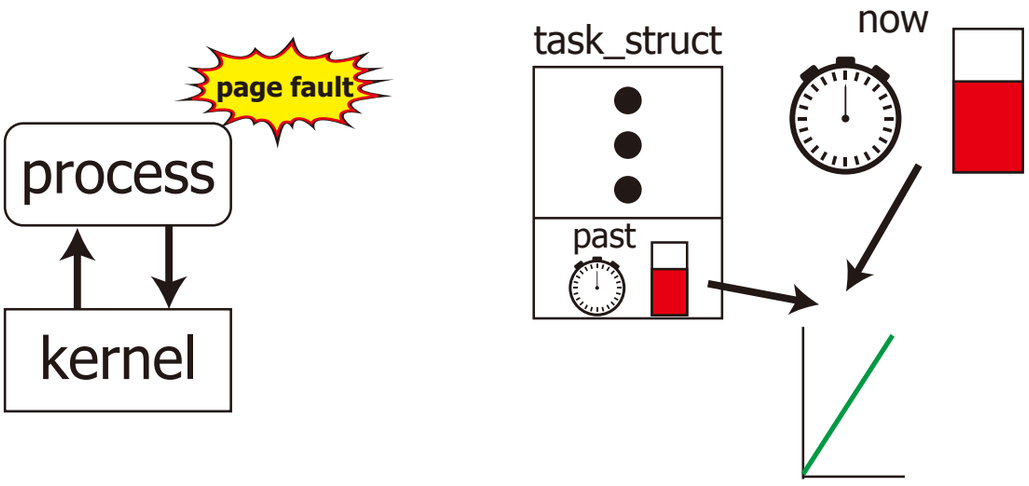

図 4.4　メモリ消費のふるまいの算出

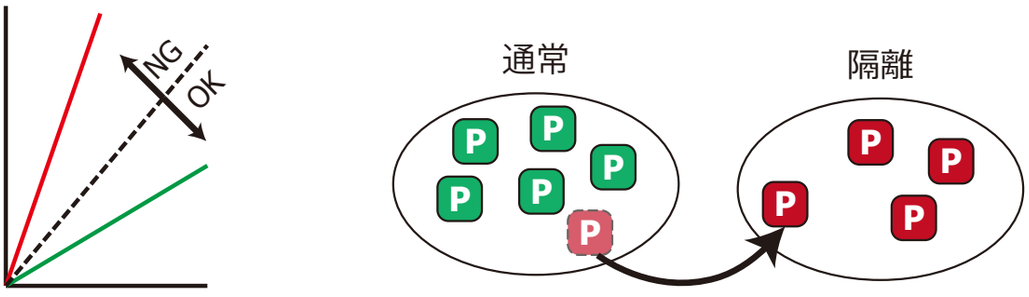

図 4.5　リソース隔離

は任意のプロセス群に対して，メモリ，CPU 割り当てなどに関するリソース管理を行う機構である．リソース制限を課す管理単位を作成し，それに制限対象となるプロセスを登録することで，プロセス群のリソース利用量の計算やリソース制限を適用することができる． この cgroups の管理単位を作成し，ふるまいが異常であるプロセス群を登録することで，リソース制限を実現する．

この cgroups の管理単位への登録は，制限対象のプロセスを検出したタイミングでは実施されない．前述したとおり，リソース制限対象の検出はページフォルト処理時に行われる．この処理は高頻度で発生することや，割り込みコンテキストで実行されるため，検出時点での cgroup の管理単位への登録処理は不適切である．制限対象のプロセスが検出されたときには，カーネルスレッドを作成し処理を遅延させる．cgroup の管理単位に登録するはそのカーネルスレッドの処理として行われる．この検出から実際の登録の間は，制限対象のプロセスによるリソースの浪費を防ぐために，そのプロセスの実行状態を TASK_UNINTERRUPTIBLE に設定し，プロセスを休止する． 作成されたカーネルスレッドで cgroup の管理単位への登録が完了した後は，プロセスの状態は TASK_RUNNING に変更され，提案手法によるリソース制限下で動作を継続する．

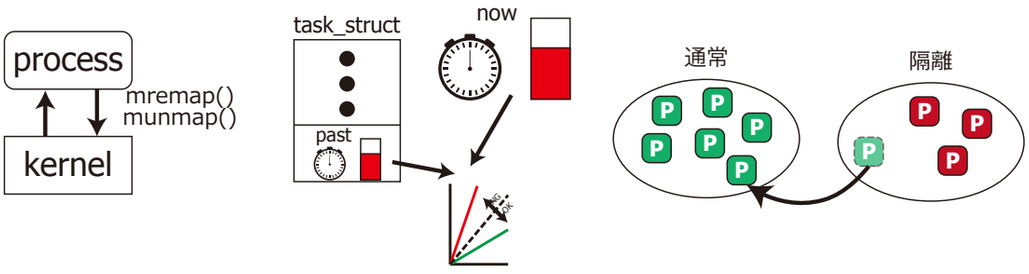

図 4.6　リソース隔離の解除

4.3.3　リソース制限の解除（メモリ解放の監視）

提案手法は，制限対象となったプロセスのメモリ解放を監視し，リソース隔離を継続するか否かを判断する．リソース消費のふるまいが正常であると判断すると，そのプロセスに対するリソース制限を解除する．この判断はリソース制限が実施された時点と，メモリ解放処理時点でのメモリ消費のふるまいを比較して行う．もし，メモリ解放処理時に，その時点でのメモリ消費のふるまいが，制限実施時のふるまいを下回っていた場合は，そのプロセスに対するリソース制限を解除する．この判断には第 4.3.2 項で述べた，リソース制限対象のプロセスを管理するデータ構造に記録されているメモリ消費のふるまいを利用する．

プロセスのメモリ解放を監視するには二つの方法がある．一つは，プロセスが OS カーネルに対してメモリ空間を返却するシステムコールを監視する方法で，もう一つが定期的に隔離したプロセスのメモリ消費のふるまいを調べる方法である．

メモリを解放するシステムコールを監視する場合は，OS カーネルの munmap システムコールおよび mremap システムコール処理に，制限対象のプロセスについて前述した判断を行う処理を追加する．この追加処理では，まず，メモリ解放のためのシステムコールを呼び出したプロセスがリソース制限の対象であるかを調べ，そうでない場合には，本来のシステムコール処理に復帰する．制限対象のプロセスが呼び出した場合は，その時点でのメモリ消費のふるまいを算出し，制限を解除するか，続行するかの判断が行われる．リソース制限を解除されるプロセスは，登録されていた制限用の cgroup の管理単位への登録を解除され，他の制限を受けていないプロセスと同様にリソースを利用することが可能になる．

一定周期で監視する方法では，カーネルタイマなどを用いて，一定周期ごとに隔離状態にあるプロセス群のメモリ消費のふるまいを調べる．前述したシステムコールを監視する方法とは，実行するきっかけが異なるだけで，内容は同一である．どちらの方法を選択するかは，具体的に対処する問題や適用するプログラムによって最適な選択が異なる．

4.3.4　リソース隔離の有効・無効の切り替え

リソース隔離を有効にするか，無効にするかの設定は，Linux カーネルが持つ，ユーザと OS カーネルのインタフェースである sysctl にパラメータとして実装する．これは種々のカーネルパラメータがファイルとして操作できるインタフェースであり，このインタフェースにリソース隔離の有効，無効を指示できるパラメータを実装した．OS カーネルのリソース隔離機構はこのパラメータが無効に設定されている場合は，いかなる場合でもリソース隔離は実施しない．一方で，メモリ消費のふるまいの計測は常に行われる．これは本節で述べているしきい値の算出のために，平常時のメモリ消費を計測する必要があるためである．

計測したメモリ消費のふるまいはカーネルメッセージとして出力する．これは計測したプロセスのプロセス番号，計測時の実メモリ使用量，計測したメモリ消費のふるまいを出力する．カーネルメッセージの出力は一般にコストが大きいため，リソース隔離の有効，無効の切り替えと同様に，メモリ消費のふるまいの出力を行うか行わないかも，sysctl のパラメータとして実装する．

4.4　まとめ

本章では, 第 1.1.2 項で述べた，Web アプリケーションプラットフォームにおけるメモリリソース利用の異常検出の問題に関して，プロセスのメモリ消費のふるまいを利用して解決することを試みた．プログラムの実行インスタンスであるプロセスは，それぞれ異なるメモリ消費のふるまいを持っている．このふるまいを利用することで，異常なメモリリソースを検出することが可能である．本章では，このふるまいに基づいた検出と，その検出したプロセスに対するリソース利用制限を組み合わせたリソース隔離を提案した．

第 5 章では，このリソース隔離を実際の問題に適用した例を示す．また，第 6 章では，その応用として，間接的にメモリの異常消費を引き起こす他のイベントのふるまいに着目した，システムにおける異常検出について述べる．

第 5 章

リソース隔離による DoS 攻撃からのシステム防御

第 4 章で述べたリソース隔離の利用例として，本章では Web アプリケーションシステムに対する DoS 攻撃からシステムを防御する手法について述べる．

Web アプリケーションはアプリケーションソフトウェアの提供形態の一つである．Web アプリケーションは Web サーバによって実行され，その結果は動的な Web ページとしてユーザへ届けられる [57]．この Web アプリケーションはオンラインサービスの提供手段として主流になりつつある．

Web アプリケーションのシステム構成としては，http サーバ，アプリケーションサーバ（AP サーバ），データベースサーバ（DB サーバ）の 3 種類のサーバによる構成が広く用いられている．ユーザからのリクエストはまず http サーバで処理され，その処理に基づいて AP サーバでサービスを提供するプログラムが動作する．AP サーバは必要に応じて DB サーバにアクセスし，必要な情報の取得や更新を実施する．リクエストに関する処理が終了すると，AP サーバはユーザへの応答を生成し，http サーバ経由で送信する．このうち AP サーバは，同時に複数のリクエストを処理するために，1 つのマスタープロセスと複数の子プロセスによって構成されている．http サーバからのリクエストを受信したマスタープロセスは，子プロセスを選択し，リクエスト処理を依頼する．

Web アプリケーションはサービス拒否攻撃（Denial of Service 攻撃：DoS 攻撃）を受ける可能性がある．DoS 攻撃はコンピュータシステムに対する攻撃の一種で，システムの正常なサービス提供を妨害することを目的とする [58]．その形態としては，大量のリクエスト送信によりネットワークを輻輳させるものや，サーバプログラムが不具合を起こすデータを意図的に送信するものなど多岐に亘る．

DoS 攻撃の中でも，Web アプリケーションの脆弱性を利用してリソースを急速に大量消費させる攻撃は特に問題である．Web アプリケーションは，その開発者やシステム管理者の意図に反して，大量のリソースを急速に消費することがある．その原因としては，アプリケーション自体のプログラミングミスや，アプリケーションフレームワーク，ライブラリ，ランタイムといった，実行環境の不具合が挙げられる．この意図しないリソース消費を引き起こすリクエストを発行することで，対象の Web アプリケーションを提供するシステムの資源を浪費させ，正当なリクエストの処理を阻害する．攻撃が発生した場合は，正当なリクエストの実行遅延や，システムの停止を招き，Web アプリケーショ

ンの運用が阻害される．大量のデータを送信してネットワークを輻輳させる量的な攻撃に対して，この脆弱性を利用したリソース浪費攻撃は少ないアクセス規模で問題を引き起こすことが可能である．Web アプリケーションに関するリソースの大量消費を引き起こすソフトウェアの不具合はこれまで多数報告されており [59–61]，また近年のソフトウェアの更新サイクルの速さを考慮すると，これからもソフトウェア脆弱性を利用した DoS 攻撃は増加していくと考えられる．

Web アプリケーションに対する DoS 攻撃を防ぐ方法としては，オペレーティングシステム（OS）レベルやアプリケーションレベルでのリソース制限機構を利用する方法，Web Application Firewall (WAF) を利用する方法がある．また，Web アプリケーションに対する DoS 攻撃を防ぐことを目的とした先行研究も存在している [62–65]．しかしながらこれらの方法には，攻撃に関する事前知識が必要である，対象としている DoS 攻撃が異なる，Web アプリケーションのクライアント側での処理が要求されるなどの問題がある．

本章では，この問題を第 4 章で述べたプロセスのリソース隔離を適用することで解決する．Web アプリケーションへのリクエストを処理するプロセスには，そのプロセス特有のメモリ消費のふるまいがある．本章での提案手法では，そのふるまいに着目して，DoS 攻撃を引き起こすリクエストを処理するプロセス検出し，リソース隔離を実施する．これにより DoS 攻撃によるリソース消費は一定以下に制限され，正常なリクエストの処理性能の低下を抑制することができる．v 提案手法は個々のリクエスト処理のメモリ消費の傾向に基づく DoS 攻撃の検出を行うため，1) 少量のアクセス規模で問題を引き起こす DoS 攻撃に対処できる．また，提案手法は攻撃の特徴などを検出に用いないため，2) 個別の攻撃の知識がなくても，問題に対応できる．さらに，提案手法はこれまでの先行手法とは異なり，DoS 攻撃と判定したリクエストを遮断したり，処理を中断しない．リソース制限を課しながらも，処理を継続することができる．そのため，3) 正常なリクエストを DoS 判定であると誤検出した場合でも，リクエストの処理を継続し，クライアントへ結果を送信することが可能である．

5.1 DoS 攻撃とメモリ消費のふるまい

本章では，Web アプリケーションの脆弱性を利用した DoS 攻撃の防止にリソース隔離を適用する．その背景として，本章が対象とする DoS 攻撃と，既存の対策手法の問題点について述べる．

5.1.1 DoS 攻撃

DoS 攻撃は，あるサービスを提供するシステムに対して，意図的にそのサービスが正常に利用できないようにする行為である [58]．DoS 攻撃は対象とするシステムや手段によってさまざまな種類があるが，本章では Web アプリケーションの脆弱性を利用して，急速なリソース消費を引き起こす DoS 攻撃を議論の対象とする．

Web アプリケーションに対する DoS 攻撃は，攻撃の対象，攻撃を引き起こすリクエストの規模によって 4 種類に分類することができる（図 5.1）．ここでは分類した 4 つの攻撃を，分類 A，分類 B，分類 C，分類 D とする．DoS 攻撃は，攻撃対象によってまず 2 つに分類することができる．1 つは Web アプリケーションを提供するプログラムを対象にしたもの，もう 1 つは，ユーザと Web アプリ

ケーションの間の通信を提供するネットワークプロトコル，http サーバなどのネットワークシステムを対象にしたものである．DoS 攻撃は，攻撃を引き起こすリクエストの規模によってさらに 2 つに分類することができる．1 つは攻撃対象に正常なリクエストを大量に送信して，Web サービスの正当な利用者の通信を阻害したり，攻撃対象のリソースを枯渇させるものである．もう 1 つは，攻撃対象の異常を引き起こすリクエストを送信し，攻撃対象を停止させたり，攻撃対象のリソースを枯渇させるものである．本章で対象にするのは，分類 B の DoS 攻撃である．

本章が対象とする DoS 攻撃の例として，コンテンツマネジメントシステムとして広く利用されている WordPress [66] に関する脆弱性を取り上げる [60] WordPress には XML-RPC [67] 経由で記事の操作を行う機能がある．WordPress で構築した Web サイト内の特定の URL に操作内容を指示する XML を送信することで，記事の取得や作成を行うことができる．WordPress の特定バージョンにはこの XML-RPC の処理に関する不具合があり，不適切な XML を送信することで Quadratic Blowup 攻撃 [68] を引き起こすことができる．Quadratic Blowup 攻撃は XML のエンティティ展開を悪用した攻撃である．DTD 内でエンティティとして長大な文字列を定義し，それを XML 本文中で大量に参照させると，XML パーサに大量のメモリ空間と CPU 時間を消費させることができる．この問題のある XML-RPC リクエストを送信することで，WordPress をホストしているシステムのリソースを急速に消費させ，WordPress に対して DoS 攻撃を引き起こすことができる．

同様の脆弱性は他の Web アプリケーションについても報告されている．ウィキエンジンの実装の 1 つである MediaWiki [69] の特定のバージョンでは，不適切なファイルのアップロードにより，メモリの大量消費を引き起こす脆弱性がある [61]．MediaWiki ではページへの添付ファイルとして SVG 画像をアップロードすることができる．SVG 画像は XML で表現されたベクタ画像データであり，その XML を処理する際に Quadratic Blowup 攻撃が起こる可能性がある．MediaWiki の特定のバージョンにはこの SVG 画像の処理に関する不具合があり，添付ファイルのアップロードにより Quadratic Blowup 攻撃を引き起こすことが可能である．

なお，Web アプリケーションやサーバソフトウェアの脆弱性を利用した DoS 攻撃としては，ソフトウェアのメモリリーク脆弱性を悪用し，長期的にメモリを浪費するものもある．本章では，この種の攻撃は対象としない．Web アプリケーションにおいては，この種の攻撃は，AP サーバの定期的に再起動することや，1 つのリクエストの処理時間に制限を設けることで十分に対処可能である．本章では，既存の方法では対策の難しい，急速なメモリ消費を伴う DoS 攻撃からの Web アプリケーションの防衛を議論する．

5.1.2　メモリの使用量に基づいたリソース制限

OS には一般的にプロセス単位で利用可能なリソースを制限する機構を持つ．また Web アプリケーションを実行するランタイムがリソース制限機構を持つこともある．これらのリソース制限機構を適切に利用すれば，Web アプリケーションによって急速なリソース消費が起こったとしても，その消費は一定以下に抑えることができる．しかしながら，メモリの使用量だけに基づいたリソース制限だけでは，本章が問題とする，急速なメモリ消費を伴う DoS 攻撃から Web アプリケーションを防御する

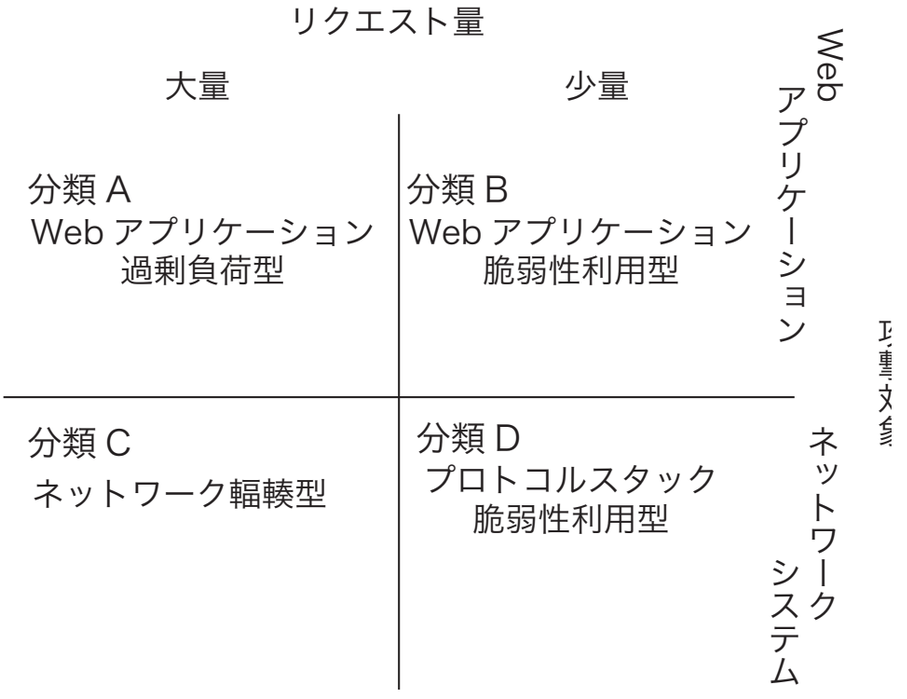

図 5.1　DoS 攻撃の分類

ことができない．

プロセスごとに使用可能なメモリ量を制限すれば，たしかに，プロセスごとのメモリの浪費は設定値以下に抑えることが可能である．しかしながら，本章の想定環境では，このメモリの浪費が同時に複数発生する可能性がある．Web アプリケーションサーバでは，リクエストを同時に複数処理するために，AP サーバの子プロセスが複数動作している．この環境下でメモリ浪費を引き起こす複数のリクエストが同時に処理されれば，システム全体としてメモリ使用量が増大し，結果としてメモリ不足を引き起こす．このメモリ不足は大量のスワップ処理を発生させ，正常なリクエストの処理性能の低下を招く．特に，メモリオーバーコミットを前提として，AP サーバのプロセス数が多めに設定されている場合は，その性能低下は顕著になる．つまり，本章が対象とする Web アプリケーションサーバにおいては，単純なメモリ使用量の制限だけでは，DoS 攻撃を防ぐことができない．

プロセスレベルのメモリ使用量制限ではなく，AP サーバ群が合計で使用できるメモリ量を規制する方法や，AP サーバのプロセス数を少なめに設定する方法も DoS 攻撃対策として考えられる．これらの方法は DoS 攻撃発生時のメモリ不足によるシステム全体の安定性確保には寄与するが，AP サーバ群のリクエスト処理性能を保証することはできない．なぜなら，その限定されたリソースを DoS 攻撃を引き起こすリクエストがメモリを浪費すれば，結果として AP サーバ全体が使用できるメモリの量が不足し，正常なリクエストの処理が阻害されるからである．このように，OS やアプリケーションランタイムが提供するメモリ使用量の制限は，個々のプロセスレベルでのメモリの浪費を抑制する効果があるが，本章が対処とするような DoS 攻撃による Web アプリケーションのリクエスト処理性

能の低下を防ぐことができない．

5.1.3 その他の対策手法

Web Application Firewall (WAF) は，Web アプリケーションのクライアントとサーバの間で，サーバに送られる通信を監視し，問題のあるリクエストを検出する機構である [57]．WAF では HTTP リクエストの内容を検査するため，Web アプリケーションの脆弱性を利用した攻撃を検出すること可能である．しかしながら，WAF はルールベースの検出を行っており，未知の脆弱性を利用した攻撃を防ぐことができない．また，検出に用いられるルールは広く利用される Web アプリケーションについて作成されるため，利用規模の小さい Web アプリケーションには対応できない．本章の提案手法では，ルールに依らない DoS 攻撃の検出を行うため，この問題は発生しない．

Web アプリケーションに対する DoS 攻撃の防止手法については，いくつかの先行研究がある [62–65]．このうち Xu らの研究 [62]，Barna らの研究 [65] は対象にしている攻撃の種類が本章とは異なっている．Ranjan らの研究 [64]，Srivatsa の研究 [63] は本章と同様に Web アプリケーションレベルのソフトウェア脆弱性を利用した DoS 攻撃の防止手法を議論している．

Ranjan らの提案手法 [64] は Web アプリケーションに対するセッションごとに，DoS 攻撃である確率を表す指標を算出し，その指標に基づいて DoS 攻撃を検出，リクエスト処理の制限を課す．その指標の算出にはクライアントからのリクエスト到達時間，リクエストを処理するためのリソース消費の傾向などを用いる．この手法はリクエスト処理のためのリソース消費に着目する点では，本章の提案手法と類似している．しかしながら，Ranjan らの方法 [64] は実際にリクエストを処理する前に，DoS 攻撃であるか否かの予測を行う．本章の提案手法では，それぞれのリクエスト処理時に実際に消費するリソース量に基づいてリソース制限を課す．そのため，より正確に DoS 攻撃の疑いのあるリクエスト処理に関するリソース利用を制限することができる．

Srivatsa [63] らの提案手法はクライアント，サーバ間でやりとりする TCP パケットに認証情報を埋め込み，正しい認証情報を持たないパケットを破棄する手法である．正しい認証情報を生成するための鍵情報はサーバから取得することができ，この発行する鍵の数を限定しておくことで，同時にアクセス可能なクライアントの数を一定以下に保つことができる．この認証情報の交換処理は HTTP レスポンスとして返される Web ページの中に JavaScript として埋め込まれており，既存の通信プロトコルを変更したり，新しいプロトコルを追加せずにクライアント，サーバ間の認証を行うことができる．Srivatsa [63] の提案手法はクライアント側での処理が必要なのに対して，本章での提案手法はサーバ側での処理のみで DoS 攻撃を防止することができる．また，Srivatsa [63] の提案手法は JavaScript が実行可能な Web ブラウザによるアクセスを前提としており，ブラウザを用いない Web API を処理するシステムなどには適用できない．これに対して本章の提案手法はクライアントの種別によらず，すべての HTTP リクエストに対応することが可能である．

5.2 提案手法

第 5.1.3 項で述べたように，既存手法では単一のリクエストでリソースの大量消費を引き起こす DoS 攻撃を防止することは難しい．この問題を解決するために，本章では第 4 章で述べたプロセスのリソース隔離の適用を試みる．

5.2.1 メモリ消費のふるまいと DoS 攻撃

メモリ消費のふるまいは，Web アプリケーションの DoS 攻撃の検出に有用である．Web アプリケーションに対するリクエストは，その実行を担うプロセスによって処理される．このプロセスのメモリ消費のふるまいには，処理する Web アプリケーションごとに固有の傾向がある．また，正常なリクエストの種類は有限であるため，そのふるまいは一定に収束する．これに対して，DoS 攻撃を引き起こすリクエストを処理するときは，定常時のふるまいとは異なるふるまいをする．この異常なメモリ消費のふるまいを検出することで，DoS 攻撃を検出することが可能である．

図 5.2, 5.3 はそれぞれ 5.1.1 節で取り上げた，WordPress と MediaWiki について，リクエスト種別ごとのメモリ消費の傾向を表したものである．それぞれの図にはテキストのみの記事（1KiB, 10KiB, 100KiB）の取得，画像を含んだ記事の取得（100KB, 1MiB, 10MiB），テキストのみの記事の投稿処理（1KiB, 10KiB, 100KiB），それぞれのリクエストを処理したときのメモリ消費の変化が示されている．図 5.4，5.5 は，それぞれ DoS 攻撃を引き起こすリクエストを処理したときのメモリ消費の変化を示している．図 5.2，5.3 と比べると，メモリ消費のふるまいが，他の正常なアクセスに比べて明らかに異なることがわかる．このふるまいの違いを利用すれば，本章が対象とする，単一のリクエストで大量のメモリ消費を発生させる DoS 攻撃を検出することが可能である．

5.2.2 ふるまいに基づいた DoS 攻撃の検出とリソース制限

本章では，第 5.2.1 項で定義したメモリ消費のふるまいに基づいて異常なメモリ消費を行うプロセスを検出し，そのリソース利用を制限する手法を提案する．これにより本章が問題とする，単一のリクエストで急速にリソースを消費する DoS 攻撃（図 5.1 における分類 B）による，正常なリクエスト処理能力の低下を防止することができる．

提案手法は，防御対象の Web アプリケーションへのリクエストを処理するプロセスのメモリ消費を監視し，正常なリクエストを処理する場合と異なるメモリ消費が起きる兆候を検出する．この検出には，プロセスのメモリ消費のふるまいを利用する．実際に使用した大きさではなく，消費する速度に着目することで，実際に大量のメモリ消費が発生する前に，その可能性を検出することができる．この検出は，防御対象の Web アプリケーションにおける DoS 攻撃でない，正常なリクエストについて，あらかじめメモリ消費のふるまいを計測しておき，そのふるまいと比較することで行う．検出されたプロセスには，リソースの利用制限が課され，DoS 攻撃によるシステムリソースの浪費を抑制する．

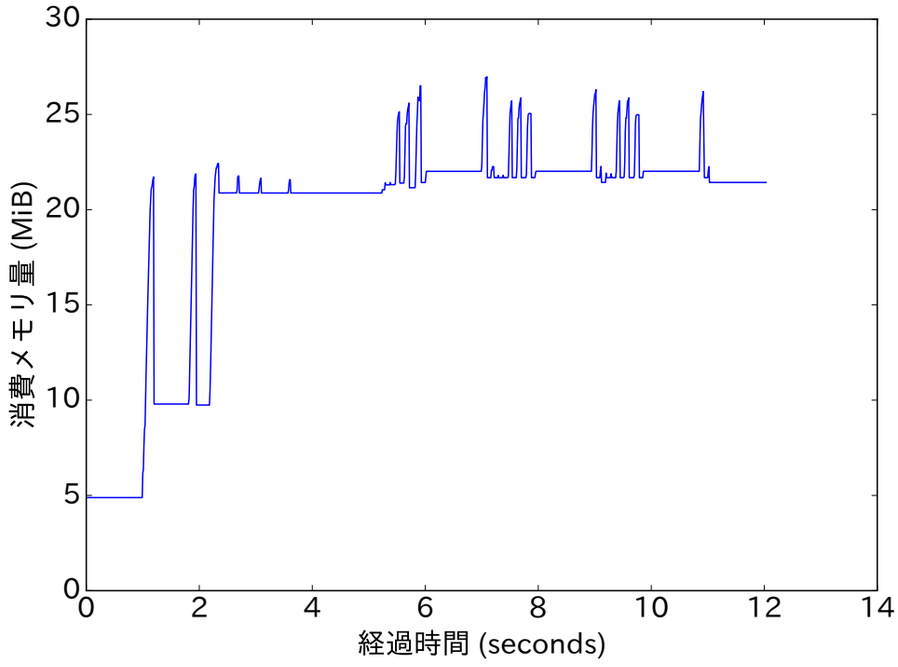

図 5.2　正常リクエスト処理時のメモリ消費のふるまい（WordPress）

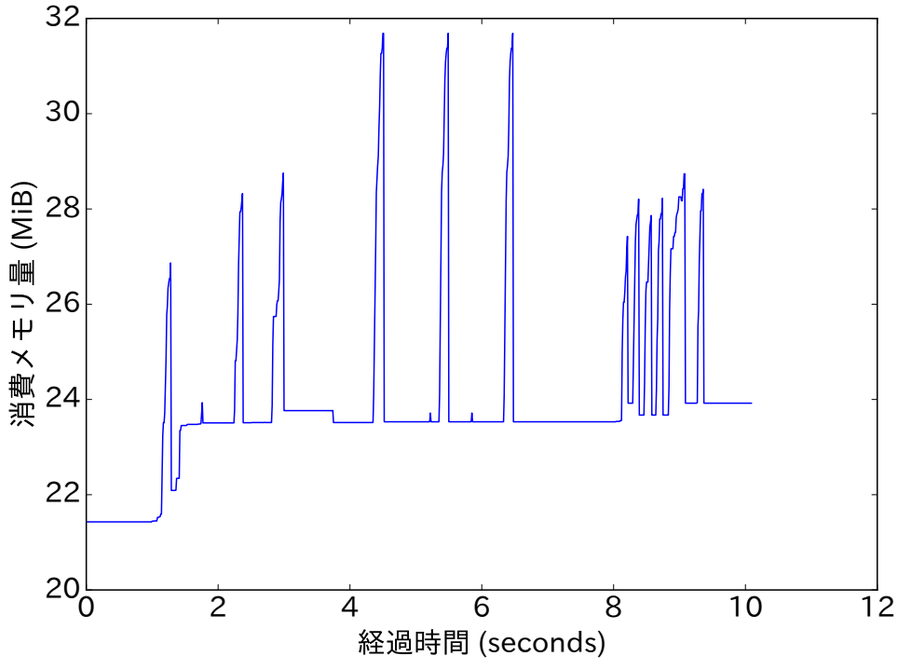

図 5.3　正常リクエスト処理時のメモリ消費のふるまい (MediaWiki)

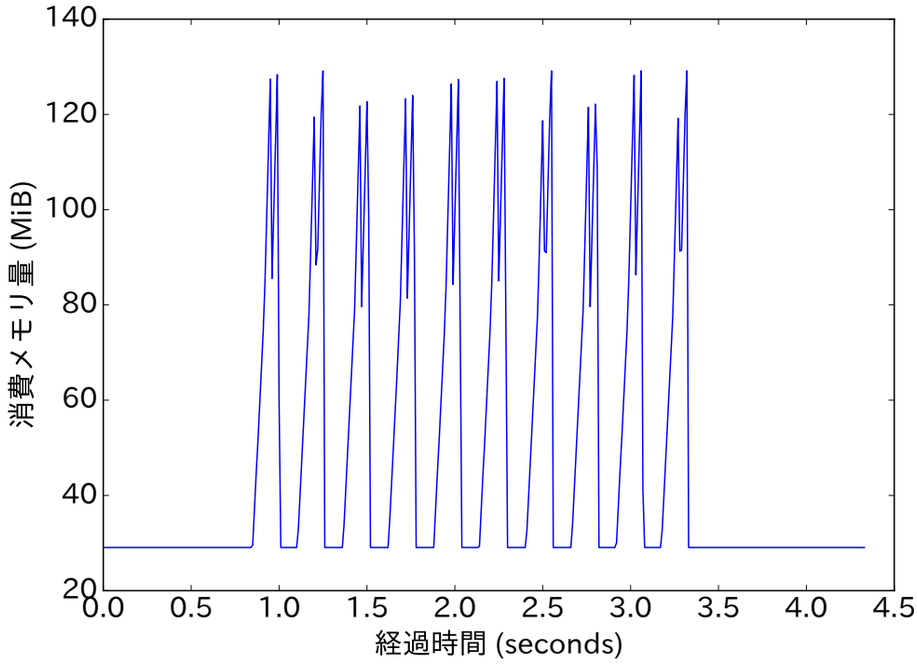

図 5.4　DoS リクエスト処理時のメモリ消費のふるまい（WordPress）

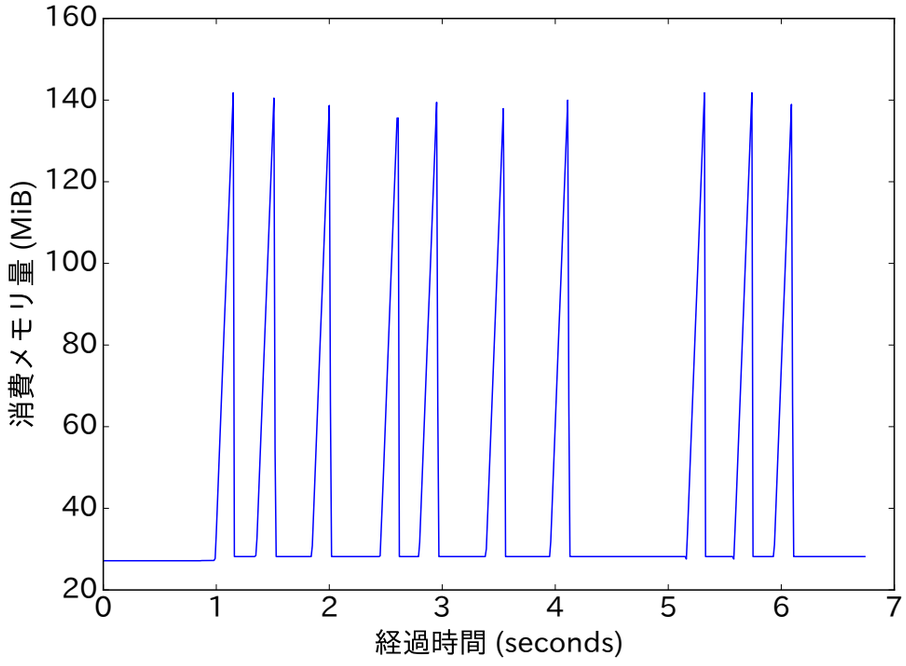

図 5.5　DoS リクエスト処理時のメモリ消費のふるまい (MediaWiki)

この提案手法には，1) リクエスト規模が小さな攻撃にも対処できる，2) 個別の攻撃に関する知識が不要である．3) 正常なリクエストを誤って DoS 攻撃であると検出した際にも，処理の継続が可能である，という 3 つの利点がある．

提案手法では，先行手法のようにアクセスパターンに基づく予測ではなく，個々のリクエスト処理ついてのメモリ消費のふるまいに基づいて攻撃を検出する．そのため，本章が対象とするような，少量のリクエスト規模で問題を引き起こす種類の DoS 攻撃に対処することが可能である．

提案手法には，防御対処の Web アプリケーションのメモリ消費のふるまいに関する知識が必要である．しかしながら，個別の攻撃に関する知識は不要である．また，ルールベースの手法が必要としている，攻撃に関する知識の継続的な更新は不要である．

提案手法や先行手法は，正常なリクエストを DoS 攻撃を引き起こすリクエストであると誤検出する可能性がある（偽陽性検出）．提案手法では，偽陽性検出の場合でも，リクエストの処理を継続することができる．先行手法では，DoS 攻撃を引き起こすリクエストであると判定した場合には，そのリクエストが遮断されたり，処理が中断される．これに対して，提案手法では，DoS 攻撃を引き起こすリクエストを検出しても，その処理はリソース制限を受けながらも継続される．そのため，応答時間が通常より大きくなりながらもリクエストは最後まで処理される．

5.2.3 しきい値の設定

第 4.3.2 項で述べたように，プロセスのリソース隔離では，パラメータ LIMIT_THRESHOLD をしきい値として，異常なメモリ消費の兆しがあるプロセスを検出する．本章での提案手法は，この検出機構を用いて，DoS 攻撃の可能性のあるリクエストを処理しているサーバープロセスを検出する．防衛対象の Web アプリケーションごとにメモリ消費のふるまいは異なるため，LIMIT_THRESHOLD は防衛対象システムごとに設定する必要がある．最適な設定値は，防衛対象の Web アプリケーションのメモリ消費のふるまいに基づいて決定する．

基準となるふるまいは，防衛対象となる Web アプリケーションにリクエストを送信し，その際のメモリ消費を計測することで得る．このために，対象 Web アプリケーションへ送信される主なリクエストを調査し，そのリクエストを再現する．このとき，仮に設定されているしきい値を超過した場合でも，リソース隔離処理は実施しない．リクエストの再現は，ユーザによって直接行うことも可能だが，計測の再現性を確保するために，Web サーバに任意のリクエストを送信する負荷生成ソフトウェアや，その計測専用のソフトウェアによって行われることが望ましい．計測したメモリ消費のふるまいの最大値を，防衛対象アプリケーションに関する最大のメモリ消費のふるまいとみなし，それに基づいてしきい値を決定する．このとき，偽陽性検出を防ぐために，計測した最大値よりも，より大きな方向へマージンを持たせる．

5.3 評価実験

提案手法の効果を検証するために実験を行った．実験の目的は，提案手法により，本章で対象とする DoS 攻撃によるシステムの性能低下を防止できることを示すことである．そのために，これまで

報告されている Web アプリケーションに関する脆弱性を利用した DoS 攻撃を再現し，提案手法の有無による，リクエスト処理性能を比較する．

また，提案手法のオーバーヘッド検証する実験も行った．提案手法を実現するためには，OS カーネルへ処理を追加する．この追加処理が Web アプリケーションの処理性能を悪化させる可能性がある．そこで，提案手法の有無によるリクエスト処理性能を比較した．

提案手法はパラメータ LIMIT_THRESHOLD によってそのふるまいが大きく変化する．最適な値を設定するためには，事前の見積もりが必要であるが，それが難しいケースや，状況の変化に合わせて最適な値が変化する可能性もある．そこで，LIMIT_THRESHOLD と提案手法の効果の変化についても実験により検証した．

5.3.1 実験環境

本章が対象とする Web アプリケーションの動作環境として，アプリケーションサーバ（AP サーバ）とデータベースサーバ（DB サーバ）を構築した．それぞれの諸元を表 5.1 に示す．また，Web アプリケーションに負荷を発生させる負荷発生器 A，負荷発生器 B を構築した．AP サーバ，DB サーバ，2 つの負荷発生器のホストサーバは単一のレイヤ 2 スイッチにより接続されている．

負荷発生器は負荷発生ソフトウェアを用いて Web アプリケーションへリクエストを送信し，AP サーバに負荷を発生させる計算機である．この計算機は仮想マシンとして実装されている．負荷発生ソフトウェアとして，Apache JMeter 3.0 [70] を採用した．負荷発生器 A と負荷発生器 B の構成は同一である．

AP サーバはクライアントからのリクエストを受け付け，処理をするためのサーバである．この処理は http リクエストの受信とレスポンスの生成を担当する http サーバプロセスと，アプリケーションの実際の処理を担当するランタイムプロセスによって提供される．ユーザから送信したリクエストは，http サーバプロセスがまず受け取り，リクエストに応じて，ランタイムプロセスに処理を要求する．ランタイムプロセスは単一のマスタープロセスと複数のスレーブプロセスから構成されており，マスタープロセスが http サーバからのリクエストをスレーブプロセスに委譲して処理を行う．本実験では，http サーバとして nginx 1.8.1 [71] を採用した．ランタイムソフトウェアには，実験対象の Web アプリケーションが依存する PHP 5.3.8 [72] に含まれる FastCGI Process Manager を採用した．

DB サーバは，Web アプリケーションが利用するデータベース管理システムが動作するサーバである．データベース管理システムとして，MariaDB [73] 15.1 を採用した．

5.3.2 実験 1：提案手法の有効性

実運用されている Web アプリケーションソフトウェアの過去のバージョンに存在したソフトウェア脆弱性を攻撃例として取り上げ，提案手法の評価を行った．取り上げたのは第 5.1.1 項で述べた，WordPress，MediaWiki に関する脆弱性である [60, 61]．それぞれの実験で，WordPress 3.9.1. MediaWiki 1.24 を使用した．

図 5.6　実験環境の概要

表 5.1　実験環境の諸元

	アプリケーションサーバ	データベースサーバ	負荷発生器
CPU	Intel Core i7-2600 (4 core)	Intel Core i7-2600 (4 core)	仮想 CPU (2 core)
RAM	2.0 GiB	2.0 GiB	2.0 GiB
Swap	4.0 GiB	4.0 GiB	5.0 GiB
OS	Linux kernel3.14.0	Linux kernel 3.10.0	Linux kernel3.10.0

表 5.2　パラメータ設定

パラメータ名	設定値	パラメータの意味
LIMIT_THRESHOLD	170 MiB/s	リソース制限を実施する閾値
MINFAULT_SAMPLE_RATE	1000	メモリ消費の監視に用いるマイナーページフォールトのサンプリング周期
MEMORY_LIMIT	256 MiB	リソース制限対象のプロセス群が利用可能なメモリ量

それぞれの Web アプリケーションに不適切なリクエストを送信すると，Web アプリケーションの処理を行うプロセスにより，メモリや CPU 時間が大量に消費される．Web アプリケーションの実行ランタイムでは利用可能なメモリ量の上限が定められており，問題のあるリクエストはこの上限までメモリを消費して停止する．そのため，上限が適切に設定されていれば，1 つの AP サーバプロセスによるメモリ消費は一定以下に抑制することができる．しかしながら，問題のあるリクエストを並行して処理する場合は，同時にメモリを浪費するプロセスの数が増え，システム全体のメモリ不足を招く．このメモリ不足により，正当なリクエストの処理が阻害される．

本実験では，負荷発生器 A, B から AP サーバに対してリクエスト負荷をかけ，その処理性能を計測した．リクエスト規模による処理性能を観察するため，リクエストを送信するスレッド数を複数変化させて計測を行った．提案手法の有無による DoS 攻撃の影響を明らかにするために，条件 A, B, C を設定した．それぞれの条件は，負荷生成器 A，B が AP サーバに対して送信するリクエストの種類，AP サーバにおける提案手法の有無の組み合わせが異なる．表 5.3 にそれぞれの条件における設定の組み合わせを示す．このうち条件 B は，従来のメモリの使用量のみに基づいてリソース制限を行う方法と同様である．

負荷生成器 A はどの条件でも正常リクエスト A を送信する．正常リクエスト A は，対象の Web アプリケーションからコンテンツを取得するリクエストとコンテンツを投稿するリクエストである．負荷生成器 A ではこの正常リクエスト A を複数のスレッドで並列に送信する．リクエスト間に休止

表 5.3　実験条件

	負荷生成器 A	負荷生成器 B	提案手法
条件 A	正常リクエスト A	正常リクエスト B	無効
条件 B	正常リクエスト A	DoS 攻撃リクエスト	無効
条件 C	正常リクエスト A	DoS 攻撃リクエスト	有効

はなく，それぞれのスレッドは可能な限り連続したリクエストを AP サーバへ送信する．

負荷生成器 B は，DoS 攻撃リクエストと正常リクエスト B を送信する．DoS 攻撃リクエストは，実験対象の脆弱性を利用して DoS 攻撃を引き起こすリクエストである．負荷発生器 B ではこの DoS 攻撃リクエストを並列数 20 で常に送信する．正常リクエスト B の内容は DoS 攻撃リクエストから，攻撃を引き起こす要素を除去したものである．つまり条件 B と C のリクエスト処理性能を比較することで，負荷発生器 B による DoS 攻撃により，リクエスト処理性能の低下がどの程度発生するのかがわかる．

パラメータ設定

第 4.3 節で述べたように，提案手法にはその動作を変化させる 3 つのパラメータがある．本実験では，実験環境の諸元に基づき表 5.2 の通り設定した．このうち，LIMIT_THRESHOLD は，リソース制限の決定に直接関与する重要なパラメータである．防衛対象の Web アプリケーションによって最適な値が異なるため，実験対象の Web アプリケーションのメモリ消費のふるまいを計測する実験を行いし，それに基づいて設定値を決定した．

実験ではそれぞれの Web アプリケーションに対して，実運用時に送信頻度が高いと推定されるリクエストを送信し，そのリクエストを処理する際の AP サーバプロセスのメモリ消費のふるまいを計測した．計測には，提案手法によるメモリ消費傾向の計測と同じ機構を用いた．WordPress, MediaWiki は共にコンテンツマネジメントシステムであり，記事の取得，投稿を行うリクエストが，全体のうち大きな部分を占めていると考えられる．よって，それぞれの Web アプリケーションについて，記事の取得を行うリクエスト，記事の投稿を行うリクエストを送信し，リクエスト処理に関するメモリ消費のふるまいを計測した．

図 5.7 はそれぞれの Web アプリケーションのメモリ消費傾向の計測結果を表したものである．青線は WordPress, 緑線は MediaWiki のメモリ消費傾向である．提案手法では，監視対象プロセスがマイナーページフォルトを一定回数起こす毎に，そのメモリ消費傾向を算出する．このグラフはその計測結果を記録し，左から計測順に並べたものである．縦軸は計測したメモリ消費傾向，横軸は計測した順番である．この結果は監視対象のプロセスのページフォルトが一定回数起きるタイミングで計測したものであり，一定の時間間隔で記録されたものではない．したがって横軸は，実際の経過時間には等間隔では対応していない．

分析の結果，WordPress についてメモリ消費傾向の最大は 165.1 MiB/s，最小は-16.4 MiB/s，平均は 50.6MiB/s であった．MediaWiki についてメモリ消費のふるまいの最大は 105.6 MiB/s，最小

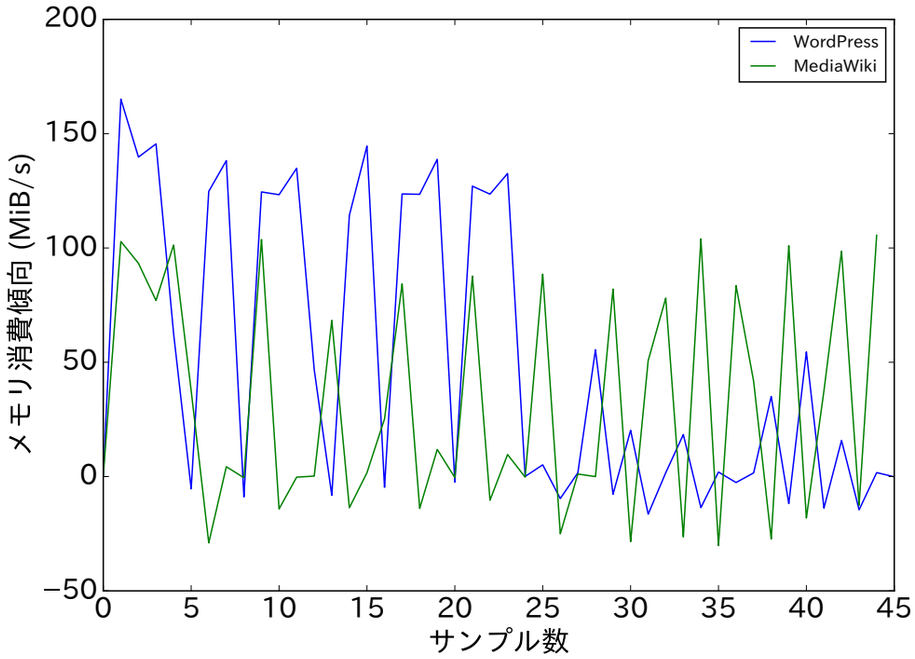

図 5.7　正常リクエスト時のメモリ消費のふるまいの変化

は-30.2 MiB/s，平均は 31.8 MiB/s であった．ここでの負の値は，使用するメモリ量の減少傾向を示している．

この結果から，2 つのアプリケーションについて通常のアクセスの範囲では最大で約 165MiB/s のメモリ消費のふるまいを示すことがわかった．本実験では，観測された最大値（165MiB）にマージンを持たせ，170 MiB/s を LIMIT_THRESHOLD に設定した．メモリ消費傾向の計測時にこの値を超えるメモリ消費傾向を示したプロセスは提案手法によるリソース制限の対象となる．

実験結果：WordPress

WordPress の脆弱性についての実験結果を図 5.8 に示す．このグラフは，それぞれの条件下での，実験対象の Web アプリケーションのリクエスト処理性能を示している．縦軸は 1 秒あたりに処理されたリクエスト数を表し，横軸は負荷生成器 A でリクエストを並行処理するスレッドの数を表している．青線，赤線，緑線はそれぞれ，条件 A，B，C における処理性能である．

条件 A（青線）と条件 B（赤線）の結果を比較すると，負荷生成器 B からの DoS 攻撃により，WordPress のリクエスト処理性能が低下していることがわかる．条件 B は条件 A と同規模の正常なリクエストを処理した際のリクエスト処理性能であり，この 2 つの条件におけるリクエスト処理性能の差を，DoS 攻撃による処理性能の低下とみなすことができる．分析の結果，条件 B のリクエスト処理性能は条件 A に対して，最大で 63.5% 低下していることがわかった．この結果から，負荷生成器 B からの DoS 攻撃により，対象の Web アプリケーションのリクエスト処理性能が低下することがわ

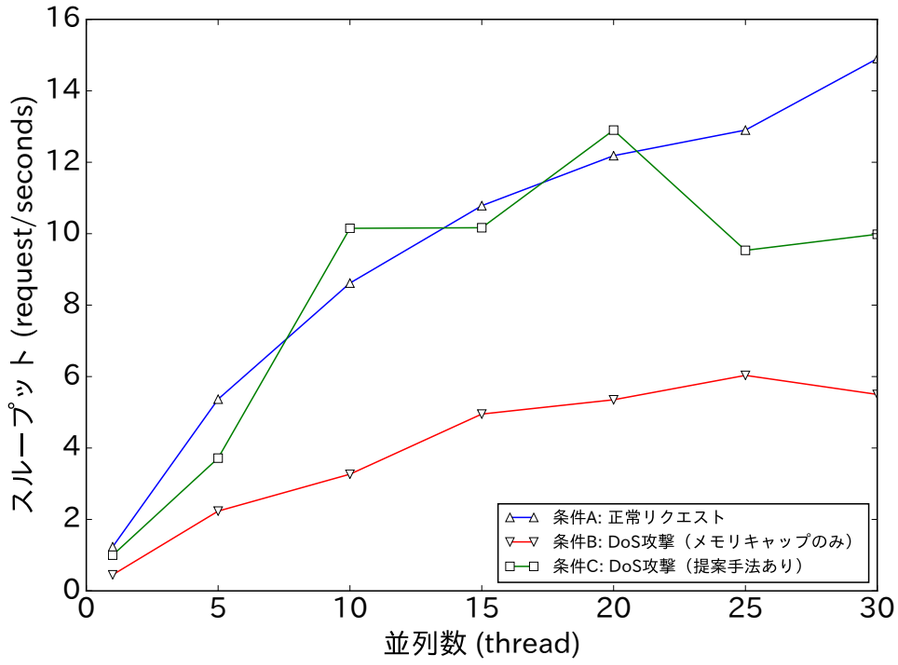

図 5.8　リクエスト処理性能 (WordPress)

かる．

条件 B（赤線）と条件 C（緑線）の結果を比較すると，提案手法により DoS 攻撃下でのリクエスト処理性能が向上していることがわかる．提案手法のない条件 B に対して，提案手法を用いる条件 C では，リクエスト処理性能は最大で 3.1 倍，最小で 1.6 倍に向上した．条件 C は AP サーバが DoS 攻撃を受ける条件 B に提案手法を適用したものであり，この性能向上は提案手法の DoS 攻撃防御効果を示している．

またこの結果からは，並列数 10，20 において，条件 C（緑線）の性能が条件 A（青線）の性能を上回っていることが確認できる．これは，条件 C では提案手法により，負荷発生器 B からのリクエストに比べて，負荷生成器 A からのリクエスト処理により多くのリソースが割り当てられるためと考えられる．条件 A では条件 B における DoS 攻撃と同じ規模の正常なリクエストが，負荷発生器 B から AP サーバに送信される．条件 B では提案手法が無効であるため，このリクエストの処理にリソース制限がかけられない．一方で，条件 C では，負荷生成器 B から DoS 攻撃リクエストが送信される．そのため，負荷発生器 B からのリクエスト処理にはリソース制限が課される．その結果として，負荷生成器 A によるリクエストの処理により多くのリソースが割り当てられ，リクエスト処理性能の向上につながったと考えられる．

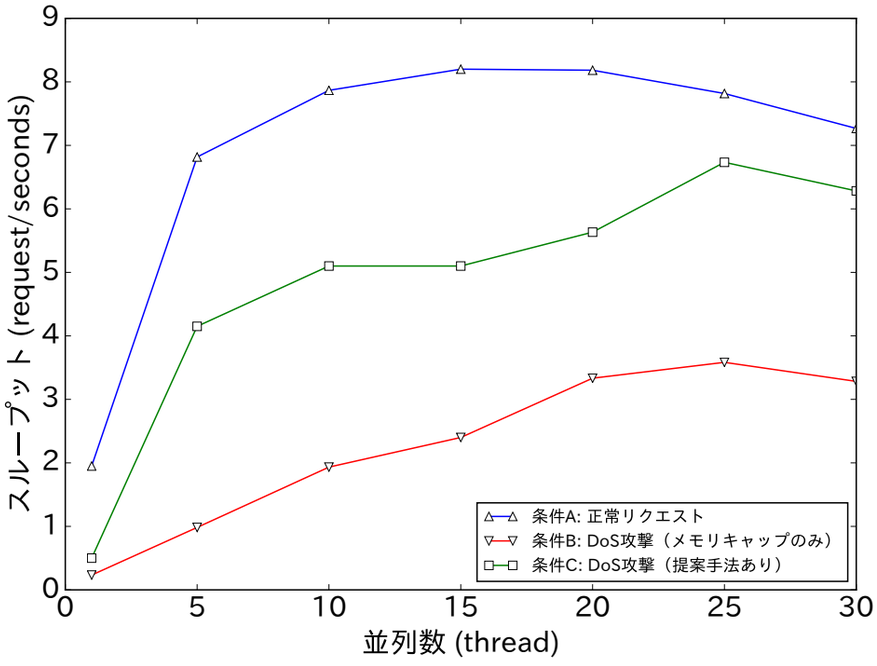

図 5.9　リクエスト処理性能（MediaWiki）

実験結果：MediaWiki

MediaWiki の脆弱性についての実験結果を図 5.9 に示す．グラフの意味については，図 5.8 に示した，WordPress の脆弱性についてと同様である．

条件 A（青線）と条件 B（赤線）の結果を比較すると，負荷生成器 B からの DoS 攻撃により，負荷生成器 B からの DoS 攻撃により，MediaWiki のリクエスト処理性能が低下していることがわかる．分析の結果，条件 B でのリクエスト処理性能は条件 A に対して，最大で 88.0% 低下していたことがわかった．WordPress での事例と同様に，負荷生成器 B からの DoS 攻撃により，対象の Web アプリケーションのリクエスト処理性能が低下することがわかる．

条件 B（赤線）と条件 C（緑線） の結果を比較すると，提案手法により DoS 攻撃下でのリクエスト処理性能が向上していることがわかる．提案手法のない条件 C に対して，提案手法を用いる条件 D では，リクエスト処理性能は最大で 4.2 倍，最小で 1.7 倍に向上した．

5.3.3　実験２：提案手法によるオーバーヘッド

提案手法を実現するためには，OS カーネルに対して処理の追加が必要である．この追加した処理が，Web アプリケーションに対するリクエスト処理を遅延させる可能性がある．そこで提案手法が Web アプリケーションに対して発生させるオーバーヘッドを評価する．

提案手法のために追加した処理の中でも，プロセスごとのメモリ消費傾向の計測する処理はリクエスト処理性能に与える影響が大きいと考えられる．この処理はプロセスへのメモリ割り当てを担うページフォルト処理に処理を追加しており，多数の呼び出しが起こる．そのため，この追加した処理がシステム全体の実行性能を悪化させる可能性は大きい．また，リソース制限をしたプロセスを一定周期で監査する処理も影響が大きいと考えられる．この処理はカーネルに短い周期の定周期タスクを追加して行うものであり，このタスクがカーネルの実行性能に悪影響を与える可能性がある．

評価は，提案手法を有効にした OS カーネル A と前述したメモリ消費傾向の計測処理，リソース制限対象の定期的な監査処理を無効にした OS カーネル B を用いて，Web アプリケーションに対するリクエスト処理性能を比較することで行った．計測対象の Web アプリケーションは実験 1 で取り上げた WordPress と MediaWiki である．これらのアプリケーションに対して DoS 攻撃を引き起こさない正常なリクエストを送信して，その処理性能をカーネル A，B 間で比較する．送信するリクエストの内容，手段は実験 1 における正常リクエスト A と同等である．

図 5.10 に計測結果を示す．赤線，青線はそれぞれカーネル A，B を用いたときの WordPress に対するリクエスト処理性能である．黄線，緑線はそれぞれカーネル A，B を用いたときの MediaWiki に対するリクエスト処理性能である．WordPress，MediaWiki，いずれの場合でも，リクエスト処理性能に大きな変化は見られなかった．分析の結果，カーネル B に対するカーネル A のリクエスト処理性能の低下は最大でも 6.0% であった．また反対に性能向上が確認できたケースもあった．この結果から，提案手法が DoS 攻撃が無い状態での Web アプリケーションの処理性能に与える影響は，非常に小さいことがわかった．

5.3.4 しきい値の変化と処理性能

LIMIT_THRESHOLD はこの提案手法の効果を左右する重要なパラメータである．このパラメータの最適値は保護対象となる Web アプリケーションごとに異なる．本実験では第 5.3.2 項で述べたとおり，対象のアプリへ想定されるリクエストを仮定してメモリ消費のふるまいを計測し，その結果に基づいて LIMIT_THRESHOLD を設定した．

この実験のように，最適な値がいつでも維持できるとは限らない．十分な事前調査ができない場合や，ソフトウェアの改修により，最適な値が変わる可能性がある．Web アプリケーションの中には，ユーザにより機能拡張できるプラグイン機構を持つものがあり，同じ Web アプリケーションでもメモリ消費のふるまいが異なることもある．

そこで，本実験で用いた LIMIT_THRESHOLD（170 MiB/sec）を基準に，そこから設定値にずれが生じると，処理性能にどのような変化を与えるのかを検証した．5.3.2 項で述べた条件 C の環境を用いて，LIMIT_THRESHOLD を変化させながら，負荷生成器 A から送信される正常リクエストの処理性能を計測した．

図 5.11 は，LIMIT_THRESHOLD を基準値から下の方向にずらしたときの，リクエスト処理性能を示している．赤の実線は LIMIT_THRESHOLD が 170 であるときのリクエスト処理性能を示しており，それ以外の破線は，LIMIT_THRESHOLD を変化させたときのリクエスト処理性能を示して

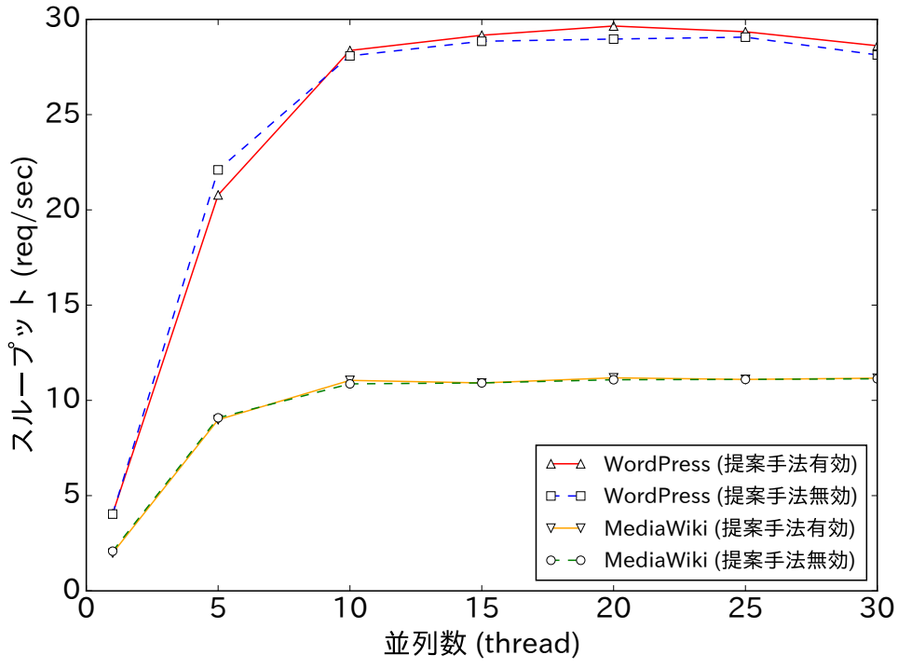

図 5.10　提案手法の有無とリクエスト処理性能の変化

いる．グラフからわかるように，LIMIT_THRESHOLD を変化させても大きな性能低下は確認できなかった．

図 5.12 は同様に LIMIT_THRESHOLD を基準値から増加させた場合のリクエスト処理性能を示している．この結果では LIMIT_THRESHOLD が 300 の場合に，全体的な性能低下が認められる．この原因は，しきい値を高くとることで，DoS 攻撃によるメモリ浪費がより防げなくなっているためと考えられる．

以上に示したように，LIMIT_THRESHOLD がその最適値から外れた場合でも，提案手法は効果を持つことがわかった．この特性は，パラメータの柔軟な決定に役立つ．

評価実験のまとめ

第 5.3.2 項で述べた実験 1 では，提案手法の，DoS 攻撃下にある Web アプリケーションのリクエスト処理能力の改善効果について検証した．WordPress, MediaWiki に関して実際に報告された脆弱性 [60, 61] を対象に，実験を行った．結果として，提案手法を用いることで，DoS 攻撃により低下したリクエスト処理性能を，最大で倍に改善することができた．

第 5.3.3 項で述べた実験 2 では，提案手法によるオーバーヘッドを評価した．実験では，提案手法が有効な OS カーネルと，提案手法の主要な処理を無効化した OS カーネルを用いて，Web アプリケーションに対する正常なリクエストの処理能力を比較した．結果として，提案手法の実装によるリクエスト処理能力の低下は最大でも 6.0% であることがわかった．

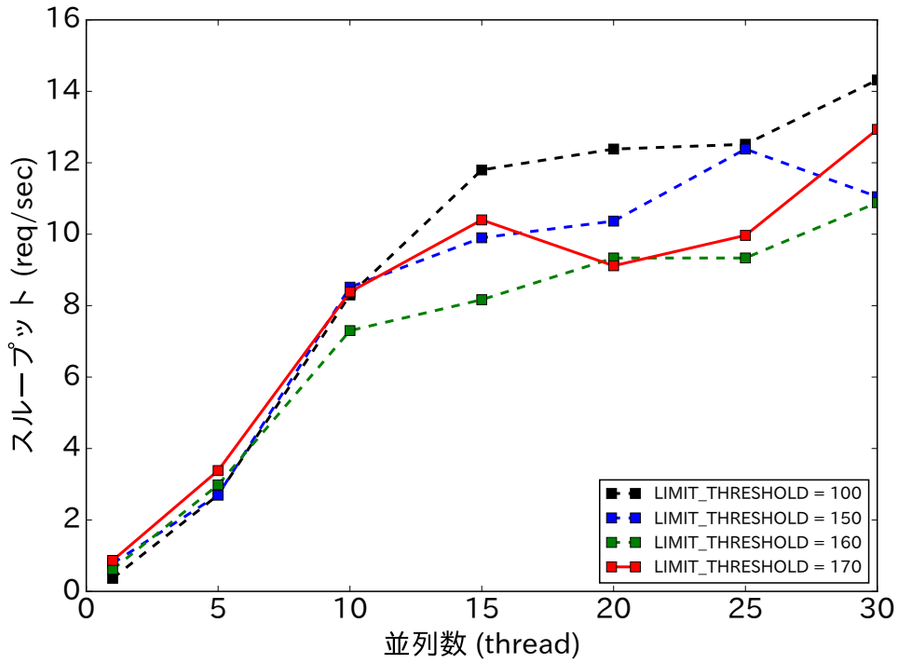

図 5.11　しきい値の変化とリクエスト処理性能（100 から 170）

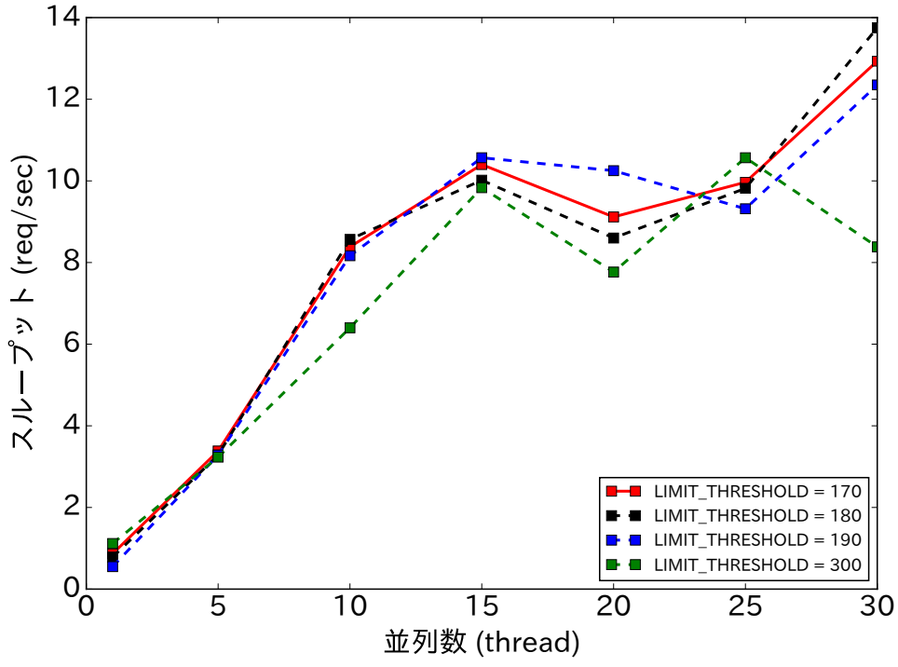

図 5.12　しきい値の変化とリクエスト処理性能（170 から 300）

以上の実験結果より，提案手法は，Web アプリケーションを DoS 攻撃によるリクエスト処理性能の低下を防止できることがわかった．また，提案手法による Web アプリケーションのリクエスト処理性能へのオーバヘッドは十分に小さいことも確認できた．

5.4 まとめ

本章では，Web アプリケーションの脆弱性を利用した DoS 攻撃の検出とその攻撃によるリクエスト処理性能低下の防止に，第 4 章で述べたプロセスのリソース隔離を適用する手法について述べた．

Web アプリケーションに対する DoS 攻撃の防止手段には，WAF や OS の機能を用いた方法や，先行研究で提案された方法があるが，いずれも本章が対象とする Web アプリケーションの脆弱性を利用した DoS 攻撃を防ぐには十分でない．プロセスのリソース消費のふるまいに基づいた異常検出とリソース隔離を行うことによって，既存手法では対処が難しかった，DoS 攻撃への対応が可能になる．

提案手法の適用例として，実際に Web アプリケーションに対する脆弱性を利用した DoS 攻撃を防止する実験を行ったところ，提案手法を用いることで，DoS 攻撃の状況下でのリクエスト処理性能の低下を小さなオーバーヘッドで抑制することができた．今後の課題としては，より広い種類の DoS 攻撃防御への適用，他の手法と提案手法組み合わせた手法の検討などが挙げられる．

第 6 章

リソース隔離による fork 爆弾攻撃からのシステム防御

第 4 章で述べたリソース隔離の利用例として，fork 爆弾攻撃からシステムを防御する方法を構築した．本章では，第 4 章や第 5 章とは異なり，間接的なメモリリソース消費のふるまいである，プロセス生成の頻度に着目する．本章ではその提案手法と評価実験について述べる．

fork 爆弾攻撃は，プロセスを高速かつ大量に生成して，対象システムのリソースを枯渇させるサービス拒否攻撃（Deniel of Service 攻撃: DoS 攻撃）[74] の 1 つである．新たなプロセスが生成されれば，そのプロセスのために新たなリソースの割り当てが起こる．このプロセス生成に伴う資源の割り当てを繰り返すことで，メモリ空間，CPU 時間，プロセス管理テーブルといった，有限リソースの枯渇が起こる．その場合，本来動作すべきプロセスが開始できない，また動作中のプロセスの動作が阻害されるといった問題が起こる．この攻撃に対処するためには，問題のあるプロセスを停止することが必要である．しかしながら，プロセス管理のためのシステムプログラムも，その動作にリソースが必要である．そのため，攻撃が発生しても有効な対策が取れず，コンピュータシステムの再起動を余儀なくされるケースもある．このように，コンピュータシステムのリソースを枯渇させ，システムの正常なサービス提供を阻害する攻撃が，fork 爆弾攻撃である．

fork 爆弾攻撃の経路は多岐にわたる．この攻撃は攻撃対象のシステムで任意のプログラムを実行可能であれば，実施することができる．そのため，攻撃対象システムに一般ユーザとしてアクセス可能であれば攻撃が可能である．また，ネットワークサービスを提供するサーバプログラムに，ネットワーク経由で任意のプログラムの実行が可能になる脆弱性があった場合には，その脆弱性を経由して，fork 爆弾攻撃が可能である．この場合は，攻撃対象のシステムに対するアクセス権限が無い状態で攻撃が可能である．サーバプログラム経由で任意のプログラムが実行可能になる脆弱性は，多数報告されている [75–80]．また存在が公表されていない脆弱性や未修正の脆弱性を利用したゼロデイ攻撃 [81] も存在するため攻撃経路を完全に断つことは難しい．システムに対する攻撃の意図がない場合でも，プログラミングミスにより，大量のプロセス生成とリソースの枯渇が起こる可能性がある [82]．また，利用者とオペレーティングシステムのインタフェースであるシェルプログラムについても，コマンドの組み合わせによっては fork 爆弾攻撃を引き起こすことが知られている [83]．

オペレーティングシステムのリソース管理機構を用いれば，fork 爆弾攻撃のような，リソース枯渇攻撃からシステムを守ることができる．しかしながら，既存のオペレーティングシステムのリソース管理機構は，fork 爆弾による攻撃を防ぐには不十分である．そこで fork 爆弾による攻撃の検出と回復についてはいくつかの提案が行われてきた [82, 84]．これらの手法は，コンピュータシステムで動作するプロセスの数や，プロセス生成の頻度に基づき，fork 爆弾攻撃を起こしているプロセス群を検出，停止することで，システムを回復させる．しかしながらこれらの手法には，fork 爆弾攻撃の原因ではないプロセスまでも，fork 爆弾攻撃を引き起こすプロセスとして検出し，誤って停止する問題がある（偽陽性検出問題）．その問題を回避するためには，fork 爆弾検出のための閾値を緩和する必要があるが，それは一方で，真に検出すべき fork 爆弾攻撃を検出できない問題を引き起こす．

本章では，この fork 爆弾攻撃の防止に第 4 章で述べたプロセスのリソース隔離を適用する．第 4 章，第 5 章では，プロセスのリソース利用のふるまいとして，メモリ消費のふるまいに着目した．それに対して本章では，間接的なメモリ消費のふるまいである，プロセス生成の速度に着目する．プロセスの生成速度に基づいて fork 爆弾攻撃を引き起こしているプロセスを検出し，そのプロセス群にリソース隔離を行うことで，fork 爆弾攻撃によるシステム性能の低下を抑制することができる．

6.1 fork 爆弾攻撃とプロセス生成のふるまい

この節では，まず提案手法の背景である fork 爆弾攻撃について述べ，既存の対策手法とその問題点について議論する．

6.1.1 fork 爆弾攻撃とその攻撃経路

fork 爆弾攻撃は，プロセスを高速かつ大量に生成して，対象システムのリソースを枯渇させるサービス拒否攻撃（Deniel of Service 攻撃: DoS 攻撃）[74] の 1 つである．

Program 6.1 は fork 爆弾攻撃を引き起こすプログラムの例である．ここでは while (true){} で示されるブロックは無限ループを表し，fork() は子プロセスを生成する手続きの呼び出しであるとする．このプログラムが動作すると，繰り返しごとに子プロセスが生成される．その生成された子プロセスは，同様に子プロセスを生成する．このように再帰的に子プロセスを生成が繰り返されると，結果として短時間に大量のプロセスが生成される．この高速かつ大量のプロセス生成はメモリ空間，CPU 資源，プロセス管理情報テーブルといった様々なリソースの枯渇を引き起こす．このリソースの枯渇が発生すると，そのシステムの安定性は損なわれ，そのシステムが提供すべきサービスが阻害される．

fork 爆弾攻撃は攻撃対象のシステムで任意のプログラムを実行可能であれば，特権権限なしに容易に実施することができる．プロセスの生成は，一般的に，管理者特権のない一般ユーザにも許可されている操作である．そのため，攻撃者が対象システムに一般ユーザとしてアクセス可能であれば，fork 爆弾による攻撃が実施可能である．また，ネットワークサービスを提供するサーバプログラムにネットワーク経由任意のプログラムを実行する脆弱性があった場合には，その脆弱性を経由して，fork 爆弾による攻撃を実施することができる．この場合は，攻撃対象のシステムに対するアクセス権限が無い状態で攻撃が可能である．サーバプログラム経由で任意のプログラムが実行可能になる脆弱

Listing 6.1　fork 爆弾攻撃を引き起こすプログラムの例

```
1 #include <unistd.h>
2 int main(void){
3     while(1){
4         fork();
5     }
6 }
```

性は，多数報告されている [75–80]．また存在が公表されていない脆弱性や未修正の脆弱性を利用したゼロデイ攻撃 [81] も存在するため攻撃経路を完全に断つことは難しい．v システムに対する攻撃の意図がない場合でも，システムの一般利用者が結果的に fork 爆弾攻撃を引き起こす可能性もある．利用者によるプログラミングの過程で，子プロセスを生成することは珍しくない．この子プロセスを生成するプログラムに，意図せずプロセス生成を繰り返す瑕疵が含まれていれば，そのプログラムは fork 爆弾攻撃を同じ状況を作り出す．また，利用者とオペレーティングシステムのインタフェースであるシェルプログラムについても，コマンドの組み合わせによっては fork 爆弾攻撃を引き起こすことが知られている [83]．問題となるプロセス生成は，プログラミングの学習過程でも起こりうる．先行研究では，大学教育でのプログラミング実習において，受講者が fork 爆弾攻撃を意図せずに起こすことが報告されている [82]．これら例のように，システムを攻撃する意図がない場合でも，ユーザのプログラミングミスなどにより，結果的に fork 爆弾攻撃が発生することがある．

6.1.2　既存手法 1: 生成プロセス数の制限

Linux には，ユーザごとに生成可能なプロセス数を制限する機構が備わっている．このプロセス数の制限により，fork 爆弾攻撃を抑制することが可能である．一般的に，1 つのシステムが管理可能なプロセスの数には上限がある．fork 爆弾攻撃は大量のプロセスを生成し，有限であるプロセス管理能力を枯渇させる．この状態では，そのシステムでは新たなプロセスを生成することができなくなり，そのシステムが本来提供すべきサービスの実行が阻害される．生成可能なプロセス数を制限することで，fork 爆弾攻撃によるプロセスの生成数を制限値に抑制することができ，プロセス管理能力の枯渇を防ぐことができる．

しかしながら，生成プロセス数の制限だけでは fork 爆弾攻撃への対策としては 2 つの点から不十分である．1 つめは，プロセス数の制限だけでは，メモリリソースの枯渇を防止できない点である．生成プロセス数の制限により，fork 爆弾によるプロセスの生成は一定に抑えることができる．しかしながら，その生成されたプロセス群は一定のメモリリソースを消費する．もし，fork 爆弾攻撃を引き起こすプログラムが，ある程度のメモリを確保するように設計されていれば，少数のプロセス生成でも，大量のメモリを消費する．例えば，ある攻撃プログラムが，メモリを 1MiB 確保するプロセスの生成を繰り返すとする．この攻撃プログラムが 1000 個のプロセスを生成した時点で，約 1GiB のメモリを占有することができる．この占有を防ぐには，ユーザが実行できるプロセス数を 1024 個以下に制限する必要がある．しかしながら，この制限はシステムの実運用を考慮すると現実的でない．この例

のように，実行可能なプロセス数の制限だけでは，fork 爆弾攻撃によるメモリリソースの枯渇を防ぐことが難しい．

２つめは，CPU リソースの浪費が防げない点である．生成プロセス数の制限により，fork 爆弾攻撃によるプロセスの増加は一定以下に抑えることが可能である．しかしながら，fork 爆弾を引き起こすプログラムが fork システムコールが失敗した場合でも引き続き fork を呼び出すようにプログラムされている場合，fork システムコールの発行は継続される．この fork システムコールの呼び出しとシステムコール処理により CPU リソースを消費する．生成プロセス数の制限のみでは，このリソース消費を防ぐことはできない．

6.1.3 既存手法 2: cgroups によるリソース制限

cgroups は Linux が実装する，プロセスのリソース管理機構である [56]．cgroups を用いることで任意のプロセス群単位でのリソース使用量の集計やリソース利用の制限を行うことができる．このリソース管理のためのプロセス群の管理単位を cgroup とする．リソース管理の単位となるプロセス群の設定は，ユーザが手動で行う他，ルールに基づいた自動構成も可能である．例えば特定のユーザもしくはユーザグループが起動したプロセスを，指定の cgroup に登録することや，特定の実行ファイルを実行したプロセスを，指定の cgroup に登録することを自動で行うことができる．

第 6.1.2 項で議論した生成プロセス数の制限と cgroups を併用することで，fork 爆弾攻撃によるプロセス数の増大を抑制しつつ，メモリリソースや CPU リソースの枯渇を防止することができる．システムで動作する全てのユーザプロセスについて，ユーザ単位，ユーザグループ単位，サービス単位で適切な cgroup を構成することで，fork 爆弾攻撃によるリソースの枯渇を防止することができる．

しかしながらこの cgroups を用いる方法には，fork 爆弾攻撃によるリソース浪費と，正当なリソースの大量消費を区別できない問題がある．システムによっては．システムのリソースを意図的に大量消費するものがある．この意図的なリソースの大量消費と fork 爆弾攻撃によるリソースの浪費を cgroups は区別しない．そのため，fork 爆弾攻撃を防ぐためのリソース制限が，システムが本来提供すべきサービスの提供を妨げる可能性がある．リソースを大量に消費するプロセスについては，リソース制限の例外を設定する方法もあるが，もし，例外としてリソース制限が緩められたプロセスに瑕疵があり，fork 爆弾攻撃を引き起こす場合は，これを防ぐことができない．

6.1.4 既存手法 3: プロセス生成頻度による fork 爆弾攻撃の検出と原因プロセスの停止

fork 爆弾攻撃が発生すると，対象システムでは短時間に大量のプロセス生成が発生する．先行研究では，この大量のプロセス生成を検知し，fork 爆弾攻撃によるリソース枯渇を防ぐ手法が提案されてきた [82, 84]．これらの手法はプロセス生成の頻度や，ユーザが実行するプロセスの数に基づいて fork 爆弾攻撃を検出する．この検出したプロセス群を停止することで，システムの不安定化の防止や，あるいは不安定化したシステムを回復させることができる．

この手法を用いる事で，fork 爆弾攻撃を検出し，その原因を除去することができる．しかしながら，

この手法には，fork 爆弾攻撃と関係のないプロセスを停止し，システムが本来提供すべきサービスの提供を阻害する問題がある．システムによっては，意図的に短時間に大量のプロセス生成が発生することがある．例えば，http サーバのプログラムが大量のリクエストに対応するために，短時間に大量の子プロセスを生成することが例として挙げられる．この子プロセスの大量生成はサービス提供のための正常な動作である．しかしながら，プロセス生成の頻度やプロセス数に基づいて，プロセスを停止する方式では，この正当なプロセス生成までもが fork 爆弾攻撃であると判定され，サーバプロセスが停止される可能性がある．

fork 爆弾攻撃と判定するための閾値を緩和することで，正当な，短時間での大量の子プロセス生成を許容し，このような偽陽性判定を回避することも可能である．しかしながらこの場合，その緩和した制限の範囲で，停止すべき fork 爆弾攻撃を引き起こすプロセスの動作も許容することになる．前述したとおり，少ない子プロセス数でリソースの枯渇を起こすことも可能であり，このような制限の緩和は偽陰性判定を招く．このように，fork 爆弾による攻撃の発生を検出し，その原因となるプロセスを停止するアプローチには，偽陽性検出により，本来のサービスの動作を阻害する問題がある．また，偽陽性検出を回避するために，fork 爆弾検出のためのパラメータを調整することは，偽陰性検出により fork 爆弾による攻撃を検出できない問題を引き起こす．

6.2 提案手法

本章では，fork 爆弾攻撃の防止にプロセスのリソース隔離を適用する．本章での提案手法は従来手法と同様に，プロセス生成の頻度に基づいて，fork 爆弾による攻撃の予兆を検出する．提案手法では，検出したプロセス群を検出時に停止しない．その代わりに，リソース隔離を行う．先行手法と異なり，プロセスを停止しないので，検出が偽陽性検出が発生した場合でも，そのプロセスは停止されない．システムの管理者や適切に権限が委譲されたユーザーは，スケジューリング禁止を解除することができる．この節ではその提案手法について述べる．

6.2.1 提案手法のねらい

提案手法のねらいは，既存手法の問題である偽陽性検出を回避しながら，fork 爆弾攻撃によるシステムの不安定化を防止することである．第 6.1.2 項から第 6.1.4 項で議論したように，既存のプロセス生成頻度に基づく fork 爆弾攻撃に基づく対策手法には，偽陽性判定により，システムが提供すべきサービスを誤って停止してしまう問題がある．これを防ぐために，提案手法では，第 4 章で述べたプロセスのリソース隔離を適用する．

また，定期的にリソース制限を課したプロセス群を監査し，システムのリソース枯渇を引き起こすふるまいを見せた場合には， そのプロセス群のスケジューリングを禁止する．一方で，リソース制限は課されたものの，実際にはシステム安定性に影響を与えないプロセス群もある．そのようなプロセス群は，反対に制限の解除を行う．

提案手法の処理の流れは，大まかに次の通りである．まず，提案手法が有効な Linux カーネルでは，fork, vfork, clone, execve といった，システムコールの呼び出しを監視し，その頻度を算出する．

もし，システムにおけるプロセス生成の頻度が閾値を超えた場合は，システムに fork 爆弾攻撃の予兆があるものと判断し，その原因となっているプロセス群に対して，リソース隔離を実施する．隔離対象となったプロセスは，その制限の範囲内で，動作を継続することができる．隔離されたプロセス群は，メモリリソースの利用状況やプロセス群に含まれるプロセスの数などを定期的に監査される．プロセス群に課せられた制限を超えてメモリリソースを使用した場合や，システムのメモリ不足の原因と判断された場合には，そのプロセス群はその後のスケジューリングから除外される．

以下ではそれぞれの処理について詳しく述べる．

6.2.2 プロセス生成速度の測定とリソース隔離

プロセスの生成は，fork, vfork, clone, execve といったシステムコールを経由して行う．提案手法では，ユーザプログラムから発行されるこれらのシステムコールをフックして，正規のシステムコールに処理を追加する．これはオペレーティングシステムが内部に持つ，システムコールハンドラテーブルを変更することによって実現する．

対象システムコールをフックした際には，そのシステムコールを発行したプロセス，そのプロセスが所属するプロセスグループ，ナノ秒単位のシステム時間を取得し，記録する．プロセス生成の記録は時系列順に並べて管理されており，この情報を用いる事で，どのような頻度でプロセス生成が行われているのかを把握することができる．通常のシステムコール処理を遅延させないために，通常は最低限の記録処理のみを行う．

一定のプロセス生成を記録すると，その記録を分析して，直近のプロセス生成の頻度を算出する．この一定のプロセス数は，パラメータ FORK_RECORD_DEPTH で指定する．このパラメータが 10 であれば，システムで起こる 10 回のプロセス生成ごとに，プロセス生成の頻度の計算を行う．

プロセス生成の頻度の計算は，FORK_RECORD_DEPTH の値を記録の先頭と最後尾の発生時間の差で除することによって求める．頻度の算出が終われば，記録していたデータは破棄され，また次のプロセス生成の記録データが記録される．

算出したプロセスの生成速度が一定の閾値を超えていた場合は，その原因となっているプロセスと，そのプロセスが所属するプロセスグループに属する全てのプロセスについてメインメモリの利用量制限を実施する．この時の閾値を FORK_SPEED_LIMIT とする．

6.2.3 隔離の解除

隔離したプロセス群があるとき，オペレーティングシステムは定期的に隔離対象のプロセス群を定期的に監査する．この間隔はパラメータ INSPECTION_INTERVAL で指定する．この監査では，隔離したプロセス群がシステムに有害な fork 爆弾であるのか，そうではないのかを見極める．

監査では，第一に課せられた制限を超えたリソースを利用していないかを調査する．cgroup によるメインメモリの制限が課されていても，システムにスワップ領域があれば，制限以上のメモリ空間を利用可能である．もちろん，設定により禁止することもできるが，その場合は，違反したプロセス群がオペレーティングシステムに強制終了される．提案手法のねらいは，fork 爆弾の疑いのあるプロセ

表 6.1　実験環境

実験環境	
CPU	Intel Core i7-2600 (4 core, 8 thread)
Operating System	Linux 3.1.0
RAM	1.0GB
Swap	2.0GB

表 6.2　実験パラメータ

パラメータ	設定値
FORK_RECORD_DEPTH	100 record
FORK_SPEED_LIMIT	600 fork/sec
MEMORY_CAP	796 MiB
INSPECTION_INTERVAL	10msec

スを強制終了せず，復帰可能な状態で停止することである．そのため，制限された主記憶を以上のメモリ空間を要求した場合に，プロセスを強制終了させるポリシーは採らない．監査時に制限を超過したプロセス群は，復帰可能な状態で停止される．これは cgroups の freezer subsystem を利用して実現される．

6.3　評価実験

本章の提案手法は，fork 爆弾攻撃を引き起こすプロセス群を検出し，そのプロセス群を停止することなく，リソース利用を制限することで fork 爆弾攻撃によるシステムの不安定化を防止する．その効果とオーバーヘッドを検証するために，3 つの実験を行った．この節ではその実験の概要と実験結果について述べる．実験は仮想マシン上に構築した環境を用いて行った．表 6.1 に実験環境の概要を示す．また提案手法に設定したパラメータを表 6.2 に示す．

6.3.1　提案手法の効果の確認

提案手法により，fork 爆弾攻撃の検出とリソース制限が実現できているか確認するために，実際に fork 爆弾攻撃によるリソース枯渇を引き起こすプログラムを実行し，システム状況の変化を観察した．

実験に用いたプログラムのソースコードを Program 6.2 に示す．実験プログラムは C 言語によって記述し，コンパイラ，リンカによって実行形式に変換して実行した．このプログラムのうち，fork() はプロセスを生成する手続きである．memset() は任意のメモリ領域を，指定されたデータを指定のサイズで埋める手続きである．usleep() はマイクロ秒単位でプログラムの実行を中断する手続きである．malloc() は任意のサイズのメモリ領域を動的に確保する手続きである．

このプログラムは，子プロセスの生成を 1 ミリ秒ごとに繰り返す．生成された子プロセスは生成された直後に，プロセス空間の特定の箇所にデータ 0x64（16 進数）を 512KiB 書き込む．Linux における子プロセス生成時のメモリコピーはコピー・オン・ライト方式で行われ，子プロセス生成時には親プロセスのメモリ内容すべてのコピーは行われない．子プロセスがメモリ内容を更新するときまで，コピーは遅延される．この memset() によるメモリの書き込みによって，子プロセスのために親プロセスから独立した領域が確保される．つまり，このプログラムによって生成された子プロセスは，それぞれ最低でも 512KiB のメモリリソースを消費する．この書き込みにより，生成された子プロセスは親プロセスとは独立したメモリ領域を確保する．

Program 6.2 を提案手法を無効にした状態で実行した．そのときのシステム全体の利用可能なメインメモリの量（空きメインメモリ）とスワップ領域の利用量の変化を図 6.1 に示す．計測は 1 秒おきに行った．Program 6.2 の実行は，空きメインメモリの測定を開始してから 3 秒後に開始した．グラフからは開始 3 秒後から空きメインメモリが急激に減少し，その後ずっと低いままであったことがわかる．分析の結果，利用可能なメインメモリ量の最小値は 52.52MiB であった．グラフからは，スワップ領域へのメモリの書き出しも多く発生していることがわかる．通常ならばスワップ領域にデータを移すことで空きメインメモリは回復することが期待されるが，この実験結果ではそうなっていない．これは Program 6.2 を実行するプロセスが，引き続きシステムにメモリを要求し続けているためだと推測される．

Program 6.2 を提案手法を有効にした状態で実行し，同様に空きメインメモリとスワップ領域の使用量を測定した．図 6.2 にその変化を示す．この場合でも測定を開始して 3 秒後から，空きメインメモリが大きく低下している．分析の結果，空きメインメモリの量は 115.02MiB まで低下していたことがわかった．しかしながら，5 秒経過後から一転増加し，14 秒目には約 425MB の空き領域を確保できていることも観察できた．これは提案手法により，fork 爆弾攻撃を起こしているプロセスを検知し，その動作を凍結していることがその理由である．提案手法が急激なプロセス生成を検出し，そのプロセス群のリソースを制限，システム全体の空きメインメモリが少なくなったことを検出した段階で，隔離の対象となっているプロセス群をプロセススケジューリングから除外する．これにより，100MiB 前後に減少した空きメインメモリを回復させることに成功している．併せてスワップ領域の利用も増えているが，これは実行を凍結されたプロセス群がページアウトされているためであると推測できる．

6.3.2 システムコール処理についてのオーバーヘッド

提案手法ではプロセス生成を検出するために，fork, vfork, clone, exec の 4 つのシステムコールをフックする．そのため，提案手法によりそれらのシステムコール処理にオーバーヘッドが加わる．この実験ではそのオーバーヘッドが実用上の問題となるか検証するために，システムコール処理時間の測定を行った．

システムコール処理の測定は Linux で標準 C ライブラリとして利用されている，glibc に含まれる fork() 関数の処理時間を測定することで実現した．提案手法がフックするシステムコールは 4 種類あ

Listing 6.2　実験プログラム

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main(void){
6      void *mem = malloc((1 << 10)*512);
7      int i, pid;
8      while(1){
9          pid = fork();
10         if(pid == 0){
11             memset(mem, 'd', (1 << 10)*512);
12         }
13         usleep(1 * 1000);
14     }
15 }
```

るが，ユーザプログラムがこれらのシステムコールを直接呼び出すことは稀であること，Linux においてはライブラリ関数の fork, vfork は clone システムコールを呼び出すこと，いずれのシステムコールのフックについても同じ処理を行うこと，以上の理由から，ライブラリ関数 fork() の処理時間を測定，比較することでオーバーヘッドを評価した．fork() の呼び出しから，結果の返却までを記録し，それを 1000 回測定した．

提案手法はプロセス生成にかかわるシステムコールを記録するため，その頻度によってオーバーヘッドが変化する可能性がある．そこで，システムコールを繰り返す頻度を変化させた 3 つの条件で測定を行った．この頻度の変化は，計測の繰り返しの合間に，適宜，間隔を開けることで実現した．特に条件 C は提案手法によりリソース制限が実施されることを意図している．このそれぞれの実験条件下で，提案手法の有効，無効を切り替えてライブラリ関数 fork() の処理時間を測定した．

- 条件 A: 1 秒間に 10 回のプロセス生成を行う条件
- 条件 B: 1 秒間に 500 回のプロセス生成を行う条件
- 条件 C: 1 秒間に 1000 回のプロセス生成を行う条件

図 6.3 から図 6.5 に，それぞれの条件でのシステムコールの処理時間を示す．このグラフは，1000 回計測したシステムコールの実行時間を，左から順番に縦方向にプロットしたものである．青のマーカーが提案手法が無効であるとき，オレンジのマーカーが提案手法が有効であるときのシステムコールの処理時間を示している．図から読み取れるように，ほとんどの場合で提案手法を有効にしても，処理時間は大きく変化していないことがわかる．しかしながら，定期的に約 4000 マイクロ秒のオーバーヘッドが起こっていることも観測できた．これは提案手法が，一定回数のプロセス生成が起こった時に，プロセス生成の頻度を計算し，リソース隔離の処理を行っていることが原因だと考えられる．条件 C については，このような大きなオーバーヘッドは 1 回しか発生していない．これは，実験プログラムが提案手法によってリソース隔離されたためである．リソース隔離をうけているプロセスにつ

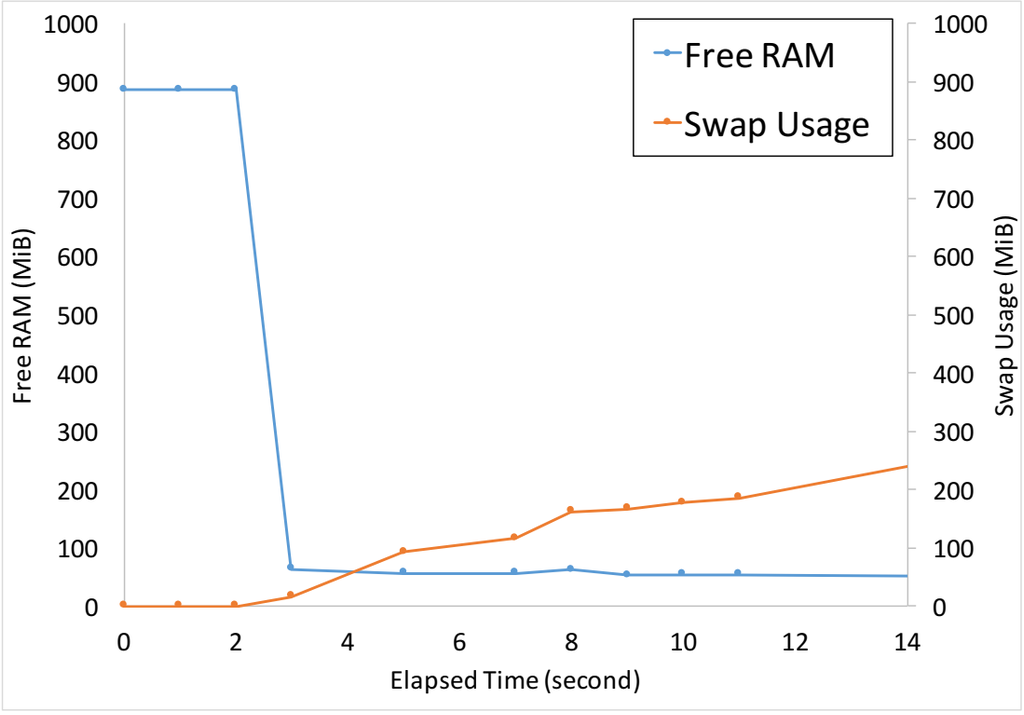

図 6.1　メモリ利用状況の変化（提案手法無効）

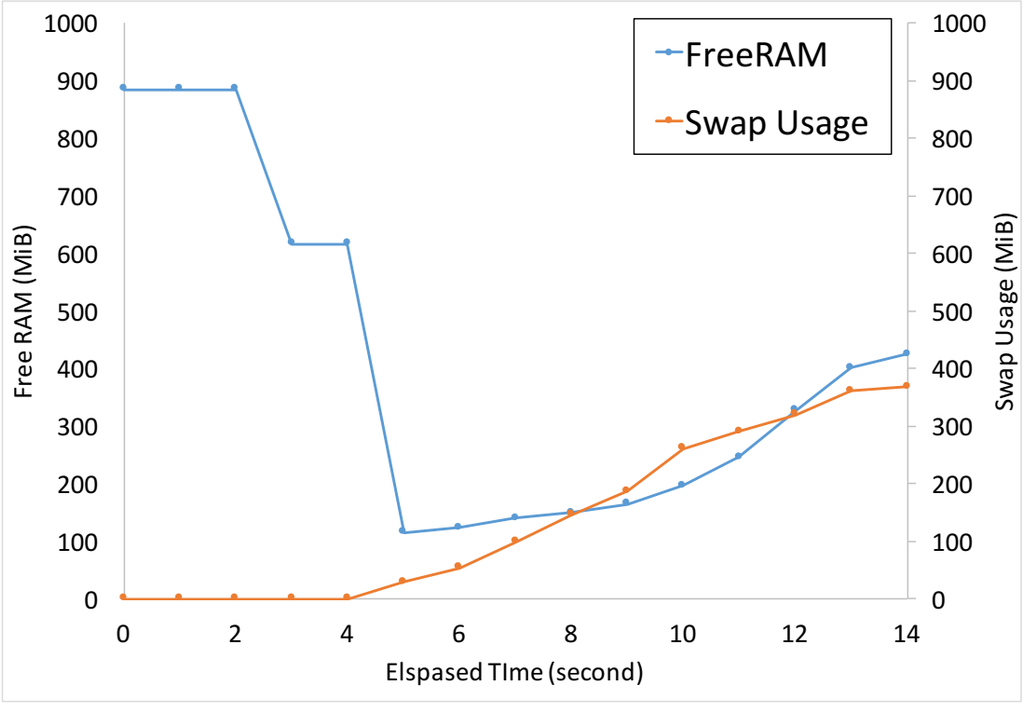

図 6.2　メモリ利用状況の変化（提案手法有効）

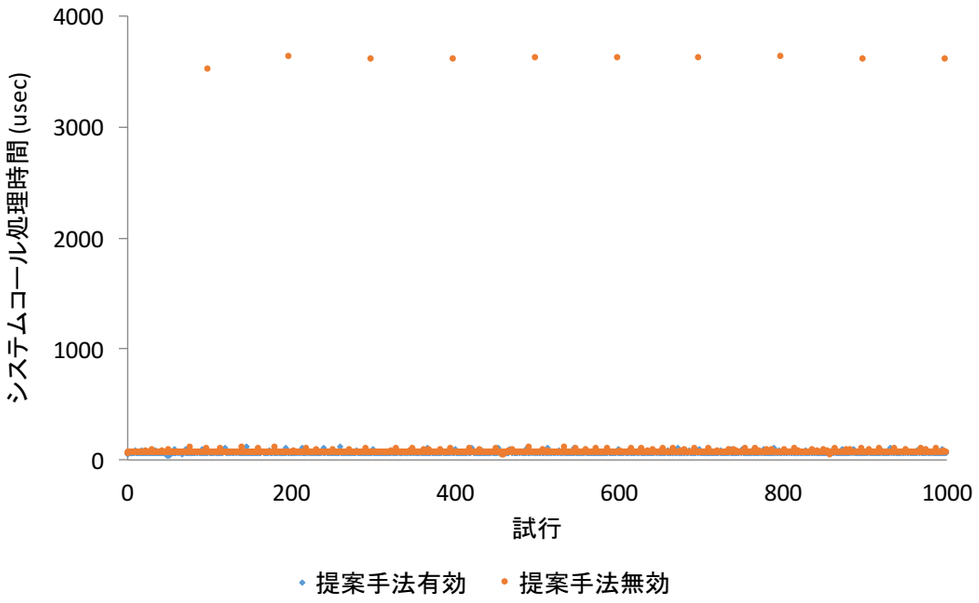

図 6.3　ライブラリ関数 fork() の処理時間（条件 A）

表 6.3　条件 A におけるライブラリ関数 fork() の処理時間 (μ sec)

	最小値	最大値	平均値	平均増加率 (%)
提案手法なし	36.53	120.72	65.43	0.00
提案手法あり	42.61	3633.53	102.10	56.03

いては，プロセス生成速度の計測から除外される．

表 6.3 から表 6.5 に，それぞの条件で測定したオーバーヘッドの処理時間を分析した結果を示す．いずれの条件でも，最小値は提案手法の有効，無効で大きな差はない．一方で，最大値については大きな差が確認された．これは図 6.3 から図 6.5 から明らかになった，プロセス生成の頻度計算とリソース隔離に伴うオーバーヘッドが原因であると考えられる．それぞれの表の最右列は，提案手法を有効にすることによって，平均処理時間がどの程度増加したかを示している．提案手法を有効にすることによって，最大で 56.6% のオーバーヘッドが発生しているが，この原因となった処理時間の増大は，それぞれの条件で 10 回発生している大きな処理時間の増大である．これは試行全体の 1% 程度であり，コンピュータシステムの用途によっては実用上の影響は軽微であると考えられる．

表 6.4　条件 B におけるライブラリ関数 fork() の処理時間 (μ sec)

	最小値	最大値	平均値	平均増加率 (%)
提案手法なし	56.57	98.12	69.67	0.00
提案手法あり	57.35	3701.95	104.90	50.56

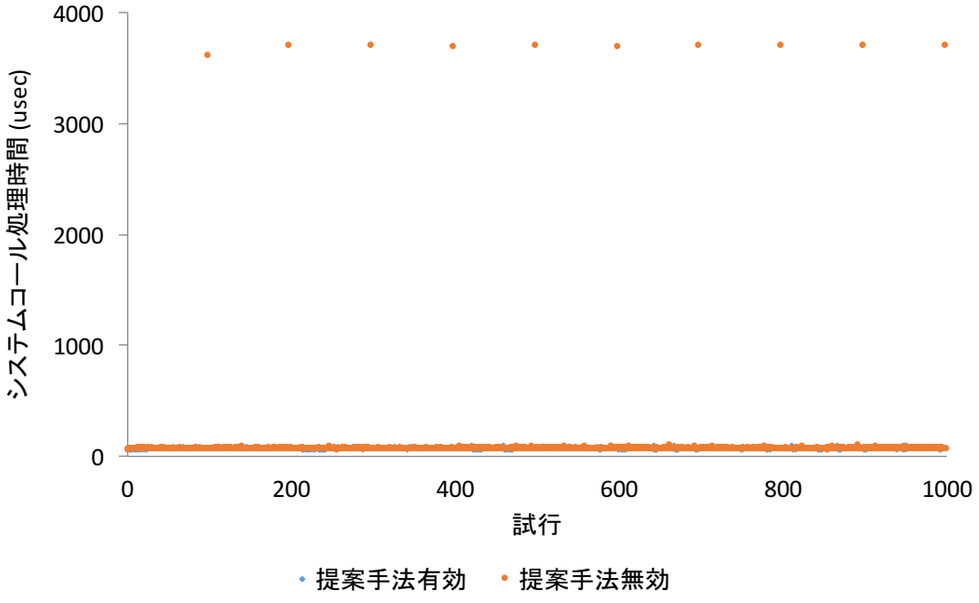

図 6.4　ライブラリ関数 fork() の処理時間（条件 B）

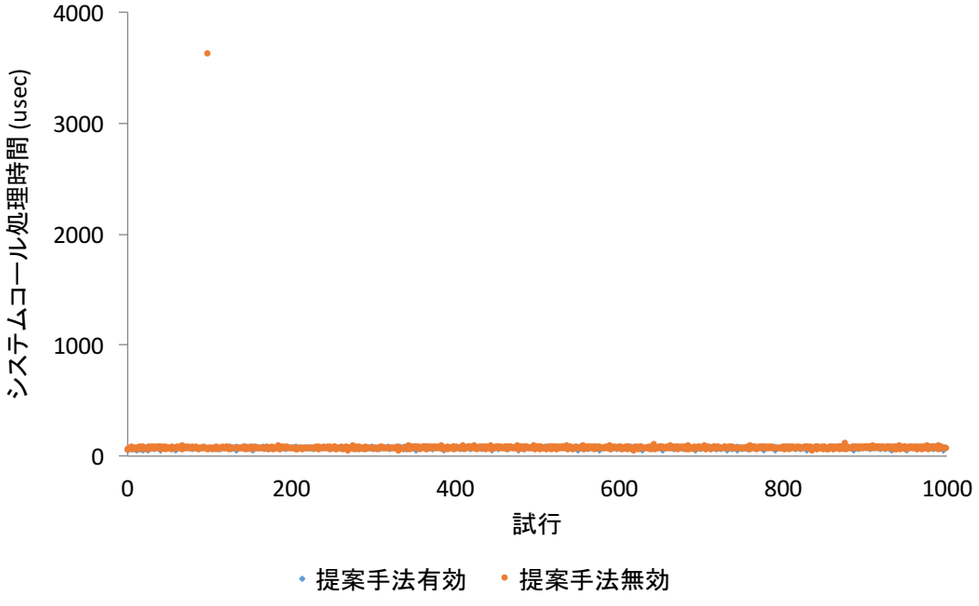

図 6.5　ライブラリ関数 fork() の処理時間（条件 C）

6.3.3 アプリケーションへの性能影響

第 6.3.2 項では，提案手法によって，定期的に 4000 マイクロ秒程度の大きなオーバーヘッドが発生することがわかった．しかしながら，そのオーバーヘッドが生じる確率は，全体の 1% 程度であることもわかった．このオーバーヘッドが，実際のアプリケーションにどの程度影響を与えるのかを検証するために，プロセスの生成を大量に発生させるワークロードを実行し，その実行時間を，提案手法が有効である場合，無効である場合で比較した．

ワークロードとしては，大規模なソフトウェアである Linux カーネルのビルドを採用した．このようなソフトウェアでは，通常，プログラムのソースコードは複数のファイルに分けて記述されており，それぞれのソースコードをコンパイルしてオブジェクトコードを生成し，それらをリンクするとで実行可能なソフトウェアを構築する．また，必要に応じてライブラリともリンクを行う．Linux 4.2.2 を対象にビルドを最小構成に設定[*1]し，そのプロセス生成数を計測したところ，95 秒間で 12629 回のプロセス生成が行われていることがわかった．図 6.6 にビルドプロセスを通した，プロセス生成数の変化を時系列で示す．このビルドプロセスを，提案手法が有効である場合，無効である場合で実行し，処理完了までの経過時間を測定した．

測定結果を表 6.6 に示す．測定はそれぞれ 10 回行い，その結果を分析した．最小値，最大値，平均値それぞれについて，大きな差は確認できなかった．第 6.3.2 項では，提案手法による大きなオーバーヘッドが確認されたが，そのオーバーヘッドが生じる確率は非常に小さく，実際のシステム性能への影響は軽微であることが確認できた．

6.4 まとめ

本章では，fork 爆弾攻撃によるリソース枯渇から，コンピュータシステムを保護する手法の提案を行った．これまでの先行研究でも，fork 爆弾攻撃の検出と原因プロセスの停止については提案されて

表 6.5 条件 C におけるライブラリ関数 fork() の処理時間 (μ sec)

	最小値	最大値	平均値	平均増加率 (%)
提案手法なし	49.93	85.14	68.25	0.00
提案手法あり	43.68	30859.24	102.71	50.49

表 6.6 Linux kernel のビルド処理時間 (sec)

	最小値	最大値	平均
提案手法なし	67.3	69.6	67.9
提案手法あり	66.7	78.0	69.2

[*1] make allnoconfig でオプションを可能な限り無効にしたビルド設定が生成可能である

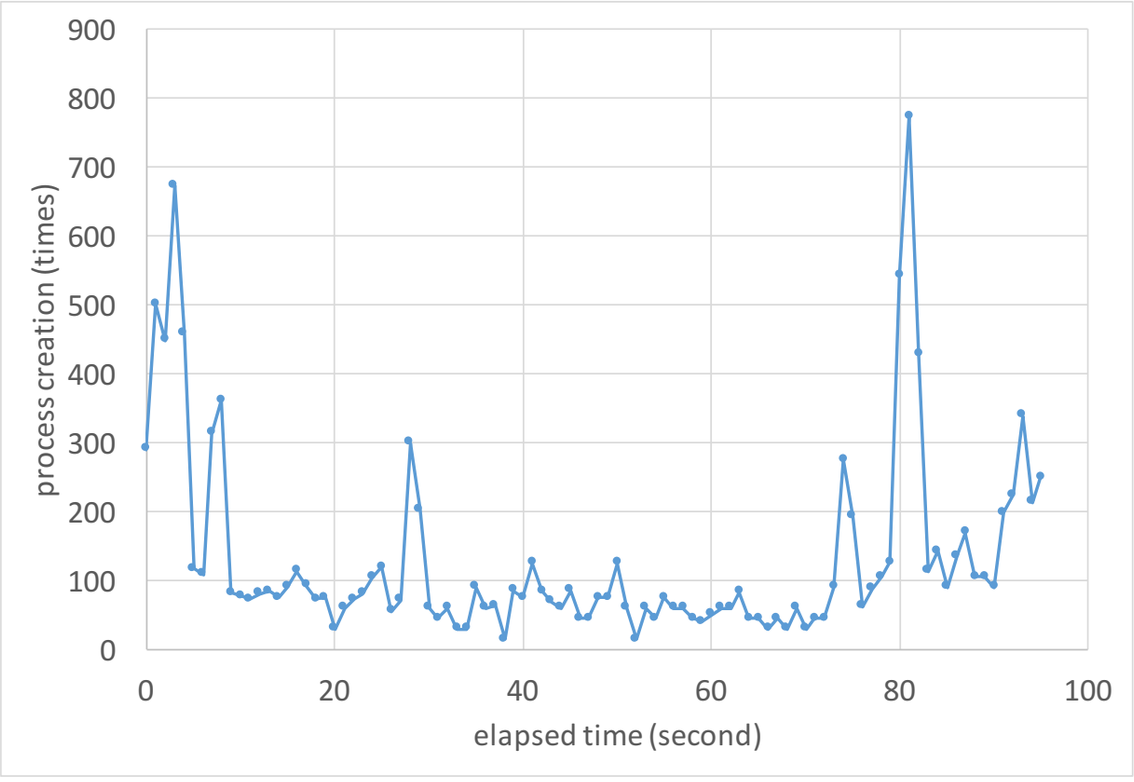

図 6.6　Linux 4.2.2 のビルドに伴うプロセス生成数の変化

きたが，それらには偽陽性検出により正常なプロセスまでも停止する危険性がある．これに対して，本章での提案手法は，fork 爆弾攻撃の予兆を捕らえても，原因プロセス群を停止しない．そのかわりに，メインメモリの利用制限を課しながら，プロセスの実行を許容する．この工夫により，偽陽性検出により，無実のプロセスが停止される危険性を低減できる．

提案手法を実装し，評価実験を行ったところ，提案手法が fork 爆弾攻撃によるリソース枯渇を防止できること，一定の間隔で大きなオーバーヘッドが生じるが，実アプリケーションを用いた評価では性能低下が起きないことがわかった．今後の課題としては，fork 爆弾検出時のリソース制限値を動的に決定すること，また状況の変化に応じて可変させることが挙げられる．現状では，リソース制限値は静的である．システムで利用可能なメインメモリの量は常に変化しており，本来ならば，システムの状況に応じて動的に決定するのが理想的である．そのための手法検討と実証実験が今後の課題である．

第 7 章

関連研究

本章では関連研究について述べる．第 3 章，第 4 章では，それぞれ異なるふるまいに着目しているため，それぞれについて議論する．

7.1　ハイブリッドメモリアーキテクチャにおけるデータ配置

第 3 章では，セマンティクスに基づいた DRAM と NVM で構成されたハイブリッドメモリアーキテクチャにおけるデータ配置について述べた．同様にハイブリッドメモリアーキテクチャにおけるデータ配置手法についてはこれまでいくつかの先行研究があるが，そのいずれもが本研究での提案手法とは異なる．

Dhiman らの研究では，NVM への書き込み回数を管理するメモリコントローラと OS の協調によって NVM と DRAM 間でページ割り当てを管理する方法を評価している [10]．Dhiman らの提案手法では，メモリコントローラのレベルで NVM に対する書き込み回数を管理することを想定しているが，このような機能は現在のメモリコントローラには未だ搭載されていない．一方で本研究での提案手法は，ソフトウェアのレベルでデータの更新頻度を管理する．そのため現行のメモリコントローラの機能を変更せずにハイブリッドメモリの管理が実現可能である．よって，Dhiman らの方法のように特別なハードウェアの実装を前提とするものに比べて，本研究での提案手法は実現が容易である．この点で先行研究と本研究での提案手法は異なる．また，データ配置の決定を実際に起こった書き込み回数に基づいて行うのか，予測を利用して行うかについても異なる．Dhiman らの方法では，全てのデータを DRAM に配置し，実際の書き込み傾向に基づいて DRAM と NVM の間でページの異動を行う．これに対して，本研究での提案手法は，データのセマンティクスに基づいてデータ更新頻度の予測を行う．そのため，実際に書き込みが発生する前に書き込みの多いデータを DRAM に，更新頻度の少ないデータを NVM に配置することが可能である．

Zhang らの研究では，データを NVM に配置し，OS のレベルで書き込みの多いページを検出して DRAM に移動させる手法を提案している [27]．この手法も Dhiman の研究と同様に，ページへの書き込み回数に基づいたデータの配置の決定を行っているため，その点で本研究とは異なる．

Ramos らの研究は Dhiman らの手法と同様にメモリコントローラと OS が協調して書き込みの多

いページを検出する [12]．この手法は書き込みの多いデータの発見に多段階のキュー構造を用いている点では，本研究での提案手法と同じである．前述した二つの手法と同様に，ふるまいに基づかないデータ配置を行っている点で本研究とは異なる．

Mogul らの研究は，プロセスのセグメントの種類に基づいて NVM と DRAM の間でデータ配置を決定する手法を提案している [11]．プロセスのメモリ領域は，その利用目的に応じて区分分けされることがある．Linux を例に挙げると，プログラムをロードする領域，スタックとして使用する領域，ヒープとして使用する領域などに区分されている．Mogul らの手法は，この区分された領域の目的に基づいて NVM と DRAM の間でデータ配置を決定する．例えば，プログラムをロードする領域は通常変更されないので，書き込みが発生しにくい領域と見なし，NVM にデータを配置する．データのセマンティクスを利用している点では，本研究での提案手法と類似している．しかしながら，Mogul らの方法がセマンティクスに対する書き込みの傾向が静的な基準で決定されるのに対し，本研究での提案手法は実行するプログラムに応じて動的に決定される．この点で本研究とは異なる．

Seok らの研究 [85]，Lee らの研究 [18] はいずれもページのアクセス履歴を用いて，write-hot なページ，write-cold なページを予測する手法である．過去のページへの書き込みアクセスのふるまいを用いた予測を行うという点では本研究と同じである．しかしながら，本研究ではデータのセマンティクスごとに異なる，書き込みアクセスのふるまいを用いて書き込み傾向を予測するため，あるデータに対して，実際に書き込みが起こる前に，それを予測することができる．その点で，先行研究とは異なっている．

Hassan らの研究 [86] は，ユーザレベルのプログラミングインタフェースレベルで，DRAM と NVM によるハイブリッドメモリを管理することを目的とした研究である．この研究での提案手法は NVM と DRAM を区別して割り当てるユーザプログラミングインタフェースと，NVMM の利用効率を調べるプロファイリングツールから構成されている．この提案手法の下で，開発者はプログラムのソースコードで NVM，DRAM を区別して割り当て，その実行ファイルをプロファイリングして，チューニングを行うことでハイブリッドメモリの効率的な利用を狙う．Hassan らの方法 [86] は，ユーザレベルでのデータの粒度でデータ配置を決定する点では本研究と同一である．しかしながら，プログラミングインタフェースのレベルでは NVM と DRAM を区別しない，実行時の状況に応じてデータ配置を動的に決定するという点において本研究での提案手法とは異なる．

7.2 Web アプリケーションにおける異常検出

第 4 章では，本研究で提案するふるまいに基づいたリソース隔離を説明し，その具体例として，第 5 章，第 6 では，Web アプリケーションに対する DoS 攻撃の検出とシステムの防御，fork 爆弾攻撃の検出とシステム防御について述べた．

プロセスのリソース隔離は，侵入検知システム（IDS: Intrusion Detection System）の一種である．IDS にはその検出の方法からシグネチャ型と異常検出型に分類できる [87]．提案手法のリソース隔離は異常検出型の IDS である．シグネチャ型の IDS はシステムに対するアクセスについて，問題のある挙動をシグネチャとして定義し，それに合致するアクセスを検出した場合に，システムに対す

る不正なアクセスを検出するものである．シグネチャ型 IDS には，正常なアクセスを偽陽性検出する可能性が低い利点があるが，一方で既知の攻撃にしか対処できない問題がある．異常検出型の IDS は，システムの平常時の状況をモデル化し，その通常の動作モデルから逸脱した動作した場合に，システムに対する不正なアクセスを検出するものである．異常検出型 IDS では，シグネチャとして定義されていない攻撃にも対処できるという点でシグネチャ型 IDS より優位であるが，一方で正当なアクセスや操作を偽陽性検出する可能性もある．本論文で述べたリソース隔離は，その異常検出型の欠点を，隔離と解放というモデルで補おうとするものである．

DoS 攻撃やそれの亜種である DDoS 攻撃を防ぐ研究はいくつかあるが，そのいずれもが本研究での提案手法とは異なる．Ranjan [64] らの研究では，Web アプリケーションに対するセッションごとに，DoS 攻撃である確率を示す指標を算出し，その指標に基づいて DoS 攻撃を検出，リクエスト処理の制限を課す．指標の算出にはクライアントからのリクエスト到達時間，リクエストを処理するためのリソース消費の傾向などを用いる．リクエスト処理のためのリソース消費に着目する点では提案手法と類似している．しかしながら，Ranjan らの方法では，実際にリクエストを処理する前に，DoS 攻撃であるか否かの予測を行うところが異なる．提案手法は実際のリクエスト処理に利用されたリソース消費に基づいて制限を課すため，より正確な検出が可能である．

Srivatsa らの研究では，クライアントの挙動を評価して，DoS 攻撃を起こさないクライアントからのリクエスト処理をより優先することで，DoS 攻撃による性能低下を防ぐ方法を提案している [63]．クライアントの挙動の追跡のために，提案手法はクライアントへのコンテンツ送信時に，管理情報とそれを管理する JavaScript のプログラムを埋め込む．埋め込まれたプログラムはクライアントの Web ブラウザで必要な処理を行い，Web サービスに情報を送信する．この手法に対し，本研究での提案手法はサーバサイドで全ての処理が完結する．よってこの点で Srivatsa の方法とは異なる．また，Srivatsa の方法はクライアントの Web ブラウザが Java Script を実行することを前提としており，REST API の提供など，Web ブラウザを用いないような Web アプリケーションでは利用できない．

7.3 fork 爆弾攻撃の防止

第 6 章では，プロセスのリソース隔離を fork 爆弾攻撃の防止に適用する手法について述べた．fork 爆弾攻撃は古くから存在する攻撃手法であり，その防止に関しては先行研究があるが，そのいずれもが本研究とは異なる．

fork 爆弾を検出して，その原因となるプロセスを停止する手法としては，Berlot の方法 [82] や Singh の方法 [84] がある．これらの方法は，対象システムにおけるプロセス生成の頻度を監視し，しきい値を超える頻度でプロセス生成が発生した場合は，fork 爆弾攻撃が発生したとみなし，原因となるプロセス群を停止する．これにより fork 爆弾攻撃によるシステム不安定化やシステム停止を防止することができる．第 6 章で述べた提案手法でも，プロセスグループごとのプロセス生成速度を利用した fork 爆弾の検出を行っている．この点で，先行研究 [82, 84] と第 6 章で述べた手法は同じである．

第 6 章で述べた方法と，先行研究 [82, 84] が異なるのは，検出したプロセスの取り扱いである．先

行研究 [82,84] では，fork 爆弾攻撃の原因となっているプロセスの動作を停止させる．一方で本研究での提案手法では，プロセスのリソース隔離を行う．第 6.1.4 項で述べたように，検出したプロセスを停止する方針には，偽陽性検出時に正当なプロセスの動作を阻害する問題がある．一方でリソース隔離を行うと，正当なプロセスのリソース利用は一時的に制限されるものの，プロセスの動作は継続され，偽陽性検出による問題を軽減することができる．この点で，先行研究 [82,84] と第 6 章で述べた方法は異なっている．

第 8 章

結論

本章では，本研究の結論を述べる．

本研究では，従来のリソース管理手法では十分に利用されてこなかったプロセスのリソース利用のふるまいに着目し，特にメモリリソースの管理に着目して，現状の問題の解決を試みた．不揮発性メモリによるメモリ階層の変化や PaaS などの新しい仮想化プラットフォームを背景として，既存の観測値のみを用いたメモリリソース管理では，メモリリソースに関する問題の解決や新しいメモリデバイスの性能を引き出すことが困難になる．この問題に対して，リソース利用のふるまいを利用して問題の解決を試みた．直近のリソース利用の状況に加えてリソース利用の経過状況から導かれるリソースの利用予測を用いる事で，問題を引き起こす事象を早い段階で検出したり，適切なメモリへのデータ配置が可能になる．このような過去の情報に基づいた予測は，これまでもプロセッサの分岐予測などに用いられてきたが，本研究ではこれをメモリリソースの割り当て管理に適用した．

本研究ではふるまいを利用したリソース管理の具体例として，ハイブリッドメモリにおけるセマンティクスに基づいたデータ配置，プロセスのリソース隔離という二つの新しいメモリ管理機構を設計，実装した．また，リソース隔離に関しては，DoS 攻撃からのシステム防衛，fork 爆弾攻撃の防止という具体的な問題への適用も行った．評価の結果，それぞれのメモリ管理機構は直近の利用状況のみに基づいたメモリリソース管理の欠点を補うことができることがわかった．

本研究ではメモリリソースに関するプロセスのふるまいに着目したが，他のリソースに関しても利用のふるまいに基づいたリソース管理が有効である可能性がある．例えば，同時マルチスレッディング (SMT) を実装したプロセッサにおいては，実行ユニットの利用のふるまいが相異なるプロセスを同時にスケジューリングすることで，実行ユニットの利用効率をより向上させることができる可能性がある．このように本研究で得られた，リソース利用のふるまいに関する知見は，他のリソース管理の研究にも有用である．

今後の課題としては，ハイブリッドメモリにおけるデータ配置手法を実際のハイブリッドアーキテクチャで検証することである．本研究の時点では，実際に利用可能なハイブリッドメモリアーキテクチャは存在していない．その登場を待って，提案手法の有効性を改めて検証する必要がある．またプロセスのリソース隔離については，より規模の大きなシステムにおける適用を検証する必要がある．

参考文献

[1] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, Vol. 9, No. 2, pp. 13:1–13:35, 2013.

[2] W W Zhuang, W Pan, B D Ulrich, J J Lee, L Stecker, A Burmaster, D R Evans, S T Hsu, M Tajiri, A Shimaoka, K Inoue, T Naka, N Awaya, A Sakiyama, Y Wang, S Q Liu, N J Wu, and A Ignatiev. Novel colossal magnetoresistive thin film nonvolatile resistance random access memory (RRAM). In *International Electron Devices Meeting. Technical Digest*, pp. 193–196, 2002.

[3] S Lai. Current status of the phase change memory and its future. In *IEEE International Electron Devices Meeting 2003*, pp. 10.1.1–10.1.4, Dec 2003.

[4] Luca Crippa and Rino Micheloni. Advanced Architectures for 3D NAND Flash Memories with Vertical Channel. In Rino Micheloni, editor, *3D Flash Memories*, pp. 167–195. Springer Netherlands, Dordrecht, 2016.

[5] JEDEC Solid State Technology Association. JEDEC Announces Support for NVDIMM Hybrid Memory Modules. `https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules`, 2015.

[6] Harbaugh Logan and Connor Deni. Diablo Technologies MEMORY1: Delivering Flash-As-System-Memory. Technical report, Storage Strategies NOW, 2015.

[7] Kishi T, Park J.W., Yoshikawa M., S. Park K., Nagase T., Sunouchi K., Kanaya H., Kim G.C., Noma K., S. Lee M., Yamamoto A., M. Rho K., Tsuchida K., J. Chung S., Y. Yi J., S. Kim H., Chun Y.S., Oyamatsu H., and J. Hong S. 4Gbit Density STT-MRAM using Perpendicular MTJ Realized with Compact Cell Structure. In *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016.

[8] J S Vetter and S Mittal. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *Computing in Science Engineering*, Vol. 17, No. 2, pp. 73–82, 2015.

[9] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. NVL-C: Static Analysis Techniques for

Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 125–136, 2016.

[10] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *Proceedings of the 46th Annual Design Automation Conference (DAC' 09)*, pp. 664–669, 2009.

[11] Jeffrey C Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS '09)*, p. 14, 2009.

[12] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*, ICS '11, pp. 85–95, New York, NY, USA, 2011. ACM.

[13] Dong-Jae Shin, Sung Kyu Park, Seong Min Kim, and Kyu Ho Park. Adaptive Page Grouping for Energy Efficiency in Hybrid PRAM-DRAM Main Memory. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pp. 395–402, 2012.

[14] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *6th Annual International Symposium on Computer Architecture (ISCA '09)*, pp. 14–23, 2009.

[15] B.C Lee, E Ipek, O Mutlu, and D Burger. Architecting Phase Change Memory as a Scalable Dram Alternative. In *36th Annual International Symposium on Computer Architecture (ISCA '09)*, pp. 2–13, 2009.

[16] Chengwen Wu, Guangyan Zhang, and Keqin Li. Rethinking Computer Architectures and Software Systems for Phase-Change Memory. *ACM Journal on Emerging Technologies in Computing*, Vol. 12, No. 4, pp. 33:1–33:40, 2016.

[17] H Yoon, J Meza, R Ausavarungnirun, R A Harding, and O Mutlu. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 337–344, 2012.

[18] S Lee, H Bahn, and S H Noh. CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures. *IEEE Transactions on Computers*, Vol. 63, No. 9, pp. 2187–2200, 2014.

[19] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S Vetter, and Sparsh Mittal. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 141–152, 2016.

[20] G Jia, G Han, J Jiang, and L Liu. Dynamic Adaptive Replacement Policy in Shared Last-Level Cache of DRAM/PCM Hybrid Memory for Big Data Storage. *IEEE Transactions on Industrial Informatics*, 2016. Early Access Article `https://doi.org/10.1109/TII.2016.`

2645941.

[21] S Bock, B R Childers, R Melhem, and D Mossé. Concurrent Migration of Multiple Pages in software-managed hybrid main memory. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 420–423, 2016.

[22] B Pourshirazi and Z Zhu. Refree: A Refresh-Free Hybrid DRAM/PCM Main Memory System. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 566–575, 2016.

[23] Z Wang, Z Gu, and Z Shao. Optimizated Allocation of Data Variables to PCM/DRAM-based Hybrid Main Memory for Real-Time Embedded Systems. *IEEE Embedded Systems Letters*, Vol. 6, No. 3, pp. 61–64, 2014.

[24] J Hu, M Xie, C Pan, C J Xue, Q Zhuge, and E H M Sha. Low Overhead Software Wear Leveling for Hybrid PCM + DRAM Main Memory on Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 23, No. 4, pp. 654–663, 2015.

[25] Y J Lin, C L Yang, H P Li, and C Y M Wang. A buffer cache architecture for smartphones with hybrid DRAM/PCM memory. In *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, pp. 1–6, 2015.

[26] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, and Jongman Kim. Designing Hybrid DRAM/PCM Main Memory Systems Utilizing Dual-Phase Compression. *ACM Transactions on Design Automation of Electorinic Systems*, Vol. 20, No. 1, pp. 11:1–11:31, 2014.

[27] Wangyuan Zhang and Tao Li. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT' 09)*, pp. 101–112, 2009.

[28] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Computer Communication Review*, Vol. 39, No. 1, pp. 50–55, 2008.

[29] Dave Durkee. Why Cloud Computing Will Never Be Free. *Queue*, Vol. 8, No. 4, pp. 20:20–20:29, 2010.

[30] Peter J Denning and Stuart C Schwartz. Properties of the Working-set Model. *Communications of the ACM*, Vol. 15, No. 3, pp. 191–198, 1972.

[31] L A Belady and C J Kuehner. Dynamic Space-sharing in Computer Systems. *Communications of the ACM*, Vol. 12, No. 5, pp. 282–288, 1969.

[32] James E Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pp. 135–148, 1981.

[33] K Ilgun, R A Kemmerer, and P A Porras. State transition analysis: a rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, pp. 181–

199, 1995.

[34] U Lindqvist and P A Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, pp. 146–161, 1999.

[35] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, Vol. 31, No. 2324, pp. 2435–2463, 1999.

[36] Brandon Craig Rhodes, James A Mahaffey, and James D Cannady. Multiple self-organizing maps for intrusion detection. In *Proceedings of the 23rd national information systems security conference*, 2000.

[37] Jun Zheng and Ming-zeng Hu. Intrusion Detection of DoS/DDoS and Probing Attacks for Web Services. In *6th International Conference on Web-Age Information Management (WAIM 2015)*, pp. 333–344. Springer Berlin Heidelberg, 2005.

[38] Ioannis A. Dimitriadis, Carlos Alberola-López, Marcos Martín-Fernández, Pablo Casaseca-de-la-Higuera, Federico Simmross-Wattenberg, and Juan Ignacio Asensio-Pérez. Anomaly Detection in Network Traffic Based on Statistical Inference and α-Stable Modeling. *IEEE Transactions on Dependable and Secure Computing*, Vol. 8, No. 4, pp. 494–509, 2011.

[39] Inho Kang, Myong K Jeong, and Dongjoon Kong. A differentiated one-class classification method with applications to intrusion detection. *Expert Systems with Applications*, Vol. 39, No. 4, pp. 3899–3905, 2012.

[40] Pedro Casas, Johan Mazel, and Philippe Owezarski. Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge. *Computer Communications*, Vol. 35, No. 7, pp. 772–783, 2012.

[41] Farzaneh Geramiraz, Amir Saman Memaripour, and Maghsoud Abbaspour. Adaptive Anomaly-Based Intrusion Detection System Using Fuzzy Controller. *I. J. Network Security*, Vol. 14, pp. 352–361, 2012.

[42] 阿部洋丈, 大山恵弘, 岡瑞起, 加藤和彦. 静的解析に基づく侵入検知システムの最適化. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 45, No. SIG03(ACS5), pp. 11–20, 2004.

[43] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pp. 156–168, Washington, DC, USA, 2001. IEEE Computer Society.

[44] S Forrest, S A Hofmeyr, A Somayaji, and T A Longstaff. A sense of self for Unix processes. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, pp. 120–128, 1996.

[45] Carla Marceau. Characterizing the Behavior of a Program Using Multiple-length N-grams. In *Proceedings of the 2000 Workshop on New Security Paradigms*, pp. 101–110, 2000.

[46] João B D Cabrera, Lundy Lewis, and Raman K Mehra. Detection and Classification of Intrusions and Faults Using Sequences of System Calls. *ACM SIGMOD Record*, Vol. 30, No. 4, pp. 25–34, 2001.

[47] T Blasing, L Batyuk, A.-D. Schmidt, S A Camtepe, and S Albayrak. An Android Application Sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 55–62, 2010.

[48] Adam G Pennington, John D Strunk, John Linwood Griffin, Craig A N Soules, Garth R Goodson, and Gregory R Ganger. Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. In *Proceedings of the 12th Conference on USENIX Security Symposium*, Vol. 12, p. 10, 2003.

[49] Grant Pannell and Helen Ashman. User Modelling for Exclusion and Anomaly Detection: A Behavioural Intrusion Detection System. In *Proceedings of the 18th International Conference on User Modeling, Adaptation, and Personalization*, pp. 207–218, 2010.

[50] Thomas J Alexandre. Biometrics on smart cards: An approach to keyboard behavioral signature. *Future Generation Computer Systems*, Vol. 13, No. 1, pp. 19–26, 1997.

[51] Li Ling, Sui Song, and C N Manikopoulos. Windows NT User Profiling for Masquerader Detection. In *2006 IEEE International Conference on Networking, Sensing and Control*, pp. 386–391, 2006.

[52] H Lieberman and C.E Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, pp. 419–420, 1983.

[53] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pp. 91–104, 2001.

[54] B Alpern and CR Attanasio. The Jalapeno virtual machine. *IBM Systems Journal*, Vol. 39, No. 1, pp. 211–238, 2000.

[55] S M Blackburn, R Garner, C Hoffman, A M Khan, K S McKinley, R Bentzur, A Diwan, D Feinberg, D Frampton, S Z Guyer, M Hirzel, A Hosking, M Jump, H Lee, J E B Moss, A Phansalkar, D Stefanović, T VanDrunen, D Von˜Dincklage, and B Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, 2006.

[56] Paul Menage. Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[57] Web Application Security Consortium. Web Security Glossary. `http://www.webappsec.org/projects/glossary/`, 2004.

[58] Handley M, Rescorla E, and IAB. Internet Denial-of-Service Considerations (RFC 4732). `http://www.rfc-editor.org/info/rfc4732`, 2006.

[59] The MITRE Corporation. CVE-2012-0789. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0789`.

[60] The MITRE Corporation. CVE-2014-5266. `http://cve.mitre.org/cgi-bin/cvename.`

cgi?name=CVE-2014-5266.

[61] The MITRE Corporation. CVE-2015-2942. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2942.

[62] Jun Xu and Wooyong Lee. Sustaining availability of Web services under distributed denial of service attacks. *IEEE Transactions on Computers*, Vol. 52, No. 2, pp. 195–208, 2003.

[63] Mudhakar Srivatsa, Arun Iyengar, Jian Yin, and Ling Liu. Mitigating Application-level Denial of Service Attacks on Web Servers: A Client-transparent Approach. *ACM Trans. Web*, Vol. 2, No. 3, pp. 15:1—-15:49, 2008.

[64] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. DDoS-shield: DDoS-resilient Scheduling to Counter Application Layer Attacks. *IEEE/ACM Transactions on Networking*, Vol. 17, No. 1, pp. 26–39, 2009.

[65] Cornel Barna, Mark Shtern, Michael Smit, Vassilios Tzerpos, and Marin Litoiu. Model-based Adaptive DoS Attack Mitigation. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '12, pp. 119–128, Piscataway, NJ, USA, 2012. IEEE Press.

[66] Matt Mullenweg, et al. WordPress.ORG. https://wordpress.org/.

[67] Dave Winer. XML-RPC Specification. http://xmlrpc.scripting.com/spec.html, 1999.

[68] Costello Roger L. XML Risks and Mitigations. https://www.mitre.org/publications/technical-papers/xml-risks-and-mitigations, 2013.

[69] Wikimedia Foundation, Inc. MediaWiki. https://www.mediawiki.org/wiki.

[70] Apache Software Foundation. Apache JMeter. http://jmeter.apache.org/.

[71] Igor Sysoev. nginx. http://nginx.org/.

[72] The PHP Group. php. http://php.net/.

[73] MariaDB Foundation. MariaDB. https://mariadb.org/.

[74] 寺田真敏. DoS 攻撃：1. DoS/DDoS 攻撃とは. 情報処理, Vol. 54, No. 5, pp. 428–435, 2013.

[75] The MITRE Corporation. CVE - CVE-2014-001. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0001.

[76] The MITRE Corporation. CVE - CVE-2014-0088. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0088.

[77] The MITRE Corporation. CVE - CVE-2014-0112. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0112.

[78] The MITRE Corporation. CVE - CVE-2014-0226. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0226.

[79] The MITRE Corporation. CVE - CVE-2015-0117. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0117.

[80] The MITRE Corporation. CVE - CVE-2015-1920. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1920.

[81] Leyla Bilge and Tudor Dumitras. Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 833–844, 2012.

[82] Michele Berlot and Janche Sang. Dealing with Process Overload Attacks in UNIX. *Information Security Journal: A Global Perspective*, Vol. 17, No. 1, pp. 33–44, 2008.

[83] David A Kennel. How Attackers Abuse Computing Systems. Technical report, Los Alamos National Lab, 2015.

[84] Rohit Singh. Fork Bomb Defuser (rexFBD). `http://rexgrep.tripod.com/rexfbd.htm`.

[85] Hyunchul Seok, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park. Efficient Page Caching Algorithm with Prediction and Migration for a Hybrid Main Memory. *SIGAPP Applied Computing Review*, Vol. 11, No. 4, pp. 38–48, 2011.

[86] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Software-managed Energy-efficient Hybrid DRAM/NVM Main Memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pp. 23:1–23:8, 2015.

[87] R A Kemmerer and G Vigna. Intrusion Detection: a Brief History and Overview. *Computer*, Vol. 35, No. 4, pp. 27–30, 2002.

謝辞

本博士学位論文は，筆者が筑波大学大学院 システム情報工学研究科 コンピュータサイエンス専攻博士後期課程在学中に取り組んだ研究をまとめたものです．本研究の遂行にあたっては，科学研究費補助金（特別研究員奨励費）「不揮発性メモリを利用可能な言語処理系の実現（研究課題 14J01818）」の助成を受けました．

本研究を進める上で熱心にご指導いただきました，本論文の主査である筑波大学システム情報系教授 追川修一先生に心より感謝を申し上げます．先生には厳しいご指導と共に多くの励ましもいただきました．ありがとうございます．

また，お忙しい中快く副査をお引き受けいただきました，筑波大学システム情報系教授 加藤和彦先生，同准教授 片岸一起先生，新城靖先生，筑波大学計算科学研究センター教授 建部修見先生に心より感謝を申し上げます．先生方との議論を通して，本研究をより深く，確かなものにすることができました．心より感謝を申し上げます．

本研究の遂行にあたっては，これまで研究と計算機工学に関する学習を共にしました，実時間組み込みアーキテクチャ研究室のメンバーにも大変お世話になりました．川崎仁氏，斎藤奨悟氏，小川直人氏，川田裕貴氏，中島基陽氏，豊田翔太氏に深く感謝します．特に斎藤氏，川田氏には，不揮発性メモリに関する研究，プロセスのリソース隔離に関する研究に関して，それぞれ重要なアイデアをいただきました．ここに深く感謝いたします．

学類生として入学して以来，今日まで学習と研究を継続できたのは，筑波大学の学生支援制度のおかげです．特に授業料に関しては最大限の支援を受けることができました．この手厚い補助制度がなければ，私が学位を取得することはできませんでした．ここに深く感謝します．

出張申請，研究費の執行，さまざまな事務手続きに関して，筑波大学大学院 システム情報工学研究科 コンピュータサイエンス専攻事務室の皆様に大変お世話になりました．手続きに関して右も左もわからない私の問い合わせに丁寧にお答えいただいたり，トラブルの解決をお手伝い頂きました．ここに深く感謝します．

博士後期課程における研究生活では，ときに困難に直面し，また孤独との闘いもありました．そんな私を支え，また内省の機会を与えてくれた音楽と，愛器である Vincent Bach 50B3O，そしてたくさんの音楽仲間に感謝します．特に「トロンボーンアンサンブルとんとろ」のメンバーとの演奏活動は，荒みがちであった私の心を落ち着かせてくれる欠かせないものでした．ここに感謝いたします．

最後に，私のわがままを許し，研究する自由を尊重してくれた妻に心よりの感謝を示します．ありがとう．

研究業績一覧

論文誌発表論文

1. 中川 岳，追川 修一：ゴミ集めの機能を応用した不揮発性メインメモリへの書き込み抑制手法の予備的評価，情報処理学会論文誌：プログラミング，Vol. 6，No. SIG 4(PRO 61)，pp. 27-37，2013.
2. Shougo Saito, Gaku Nakagawa, Shuichi Oikawa: Language Runtime Support for Non-Volatile Main Memory Management, ICIC Express Letters, Vol.8, No.1, pp. 137-144, 2014.
3. Hirotaka Kawata, Gaku Nakagawa, Shuichi Oikawa: Using DRAM as Cache for Non-Volatile Main Memory Swapping, International Journal of Software Innovation, vol.4, No.1, pp. 61-71, 2016.
4. 中川 岳，追川 修一：Web アプリケーションサーバにおけるプロセスのふるまいに基づいた DoS 攻撃の防御手法，情報処理学会論文誌：コンピューティングシステム，採録決定.

国際会議発表

1. Gaku Nakagawa, Shuichi Oikawa: Consideration on an Operating System for Heterogeneous Multi-Core Architecture, 2012 International Conference on Future Information Technology and Management Science & Engineering, pp. 208-212, 2012.
2. Shuichi Oikawa, Gaku Nakagawa, Naoto Ogawa, Shougo Saito: Extending RTOS Functionalities: an Approach for Embedded Heterogeneous Multi-Core Systems, 7th International Conference on Computing in the Global Information Technology (ICCGI 2012), pp. 136-139, 2012.
3. Gaku Nakagawa, Shuichi Oikawa: “Consideration on a Power-Saving and Reliable Embedded System”, 1st IEEE Global Conference on Consumer Electronics (GCCE 2012), pp. 516-519, 2012.
4. Gaku Nakagawa, Shuichi Oikawa: An Architecture of Operating System utilizing Non-volatile Main Memory and Heterogeneous Multi-Core, 12th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2013), pp. 559-563, 2013.

5. Gaku Nakagawa, Shuichi Oikawa: Preliminary Analysis of A Write Reduction Method for Non-Volatile Main Memory on Jikes RVM, 4th International Workshop on Advances in Networking and Computing, 1st International Symposium on Computing and Networking (CANDAR 2013), pp. 597-601, 2013.
6. Gaku Nakagawa, Shuichi Oikawa: Language Runtime Support for NVM/DRAM Hybrid Main Memory, IEEE Symposium on Low-Power and High-Speed Chips (COOL Chips XVII), 3 pages, 2014.
7. Gaku Nakagawa, Shuichi Oikawa: An Analysis of The Relationship between A Write Access Reduction Method for NVM/DRAM Hybrid Memory with Programming Language Runtime Support and Execution Policies of Garbage Collection, 12th International Conference on Software Engineering Research, Management and Applications (SERA 2014), pp. 597-603, 2014.
8. Gaku Nakagawa, Shuichi Oikawa: NVM/DRAM Hybrid Memory Management with Language Runtime Support via MRW Queue, 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2015), pp. 357-362, 2015.
9. Gaku Nakagawa, Shuichi Oikawa: Multi-level Queue NVM/DRAM Hybrid Memory Management with Language Runtime Support, 2015 Research in Adaptive and Convergent Systems (RACS 2015), pp.437-442, 2015.
10. Gaku Nakagawa, Hirotaka Kawata, Shuichi Oikawa: On-the-fly Process Resource Quarantine for System Stabilization, The 15th IEEE International Conference on Computer and Information Technology (CIT-2015), pp.517-524, 2015.
11. Hirotaka Kawata, Takahiro Hirofuchi, Ryousei Takano, Gaku Nakagawa, Shuichi Oikawa. Modeling Energy Consumption of Memory Systems, 6th International Workshop on Advances in Networking and Computing, The Third International Symposium on Computing and Networking (CANDAR 2015), pp. 601-603, 2015. (poster session)
12. Gaku Nakagawa, Hirotaka Kawata, Shuichi Oikawa: Out of memory prevention based on memory allocation rate, 6th International Workshop on Advances in Networking and Computing, The Third International Symposium on Computing and Networking (CANDAR 2015), pp. 567-570, 2015.
13. Gaku Nakagawa and Shuichi Oikawa: Fork Bomb Attack Mitigation by Process Resource Quarantine, 7th International Workshop on Advances in Networking and Computing, The Fourth International Symposium on Computing and Networking (CANDAR 2016), pp. 691-695, 2016.
14. Gaku Nakagawa and Shuichi Oikawa:Behavior-based Memory Resource Management for Container-based Virtualization, 3rd International Conference on Computational Science/Intelligence & Applied Informatics (CSII 2016), pp.213-217, 2016.

15. Shuichi Oikawa and Gaku Nakagawa: Memory Interface Simplifies Storage Virtualization, The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, pp. 1-3, 2017.
16. Gaku Nakagawa and Shuichi Oikawa: Data Placement Based on Data Semantics for NVDIMM/DRAM Hybrid Memory Architecture, The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, pp. 99-101, 2017.

その他の研究業績

1. 中川 岳，追川 修一：ヘテロジニアスマルチコアを統合管理するオペレーティングシステム構成法の検討，第 120 回 システムソフトウェアとオペレーティング・システム研究会，2012 年 2 月.
2. 中川 岳, 追川 修一：ゴミ集めの機能を応用した不揮発性メインメモリへの書き込み抑制手法の実装と効率改善，第 99 回プログラミング研究会，2014 年 6 月.
3. 中川 岳，追川 修一：次世代不揮発性メモリによる主記憶拡張のためのソフトウェアサポート，第 12 回 ディペンタブルシステムワークショップ (DSW2014)，2014 年 12 月.
4. 中川 岳，追川 修一：コンテナ型仮想化の機能を応用したページング方式，第 132 回 システムソフトウェアとオペレーティング・システム研究会，2015 年 2 月.
5. 中川 岳，川田 裕貴，追川 修一：プロセスの動的リソース隔離によるシステム安定化手法，第 134 回 システムソフトウェアとオペレーティング・システム研究会，2015 年 8 月.
6. 中川 岳，川田 裕貴，追川 修一：プロセスのリソース隔離による Fork 爆弾攻撃の防止手法，コンピュータシステム・シンポジウム論文集（ComSys2015)， pp. 16-23，2015.
7. 中川 岳：プロセスのふるまいに基づいたリソース管理，第 13 回 ディペンタブルシステムワークショップ (DSW2015)，2015 年 12 月.
8. 中川 岳，川田 裕貴，追川 修一：メモリ消費のふるまいに基づくメインメモリ管理手法，第 136 回 システムソフトウェアとオペレーティング・システム研究会，2016 年 3 月.